BUILDING DATA-CENTRIC SECURITY MECHANISMS FOR WEB APPLICATIONS

A Thesis
Presented to
The Academic Faculty

by

Yogesh Mundada

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
May 2016

BUILDING DATA-CENTRIC SECURITY MECHANISMS FOR WEB APPLICATIONS

Approved by:

Professor Nicholas G. Feamster,
Advisor
Computer Science
Princeton University

Professor Wenke Lee
School of Computer Science
Georgia Institute of Technology

Professor Mostafa Ammar
School of Computer Science
Georgia Institute of Technology

Professor Arvind Narayanan
Computer Science
Princeton University

Professor Mustaque Ahamad
Scool of Computer Science
Georgia Institute of Technology

Date Approved: 4 April 2016

To Meera, my mother

ACKNOWLEDGEMENTS

I sincerely want to thank Professor Nick Feamster, my adviser, for his support and direction, and for giving me complete freedom throughout this undertaking.

I also want to thank my co-authors and colleagues for many intense brainstorming sessions and research discussions, at the end of which, I always gained useful knowledge. Mainly, I would like to mention, Balachander Krishnamurthy, Anirudh Ramachandran, Dave Levin, Saikat Guha, and Srikanth Kandula. Additionally, I'd like to thank my thesis committee members, who helped me polish my thesis and my defense presentation. I am grateful to Professor Mostafa Ammar, Professor Mustaque Ahamad, Professor Wenke Lee and Professor Arvind Narayanan for their guidance and patience. I also want to thank Professor Venkateswaran for his kind words during stressful times.

Achieving a PhD is a tough journey. I would not have made until the end without backing from my brother Dinesh and all of my friends: Bilal Anwer, Maria Konte, Ilias Fountalis, Aemen Lodhi, Bhanu Chandra Vattikonda, Samantha Lo, Karim Habak, Tyler Winegar, Eddie Sarkisov, Paul Medeiros, Martin Henderson, Girish, Ali, Yogesh Chavan, Milind, and Angi.

Lastly, I could always count on Shannon Croft to put me back in the right mindset when things got gloomy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Data loss from web applications at different points of compromise has become a major liability in recent years. Existing security guidelines, policies, and tools fail often, ostensibly for reasons stemming from blatant disregard of common practice to subtle exploits originating from complex interactions between components.

Current security mechanisms focus on "how to stop illicit data transfer" (i.e., the "syntax"), and many tools achieve that goal in principle. Yet, the practice of securing data additionally depends on allowing administrators to clearly specify "what data should be secured" (i.e., the "semantics"). Currently, translation from "security semantics" to "security syntax" is manual, time-consuming, and ad hoc. Even a slight oversight in the translation process could render the entire system insecure. Security semantics frequently need modifications due to changes in various external factors such as policy changes, user reclassification, and even code refactoring.

This dissertation hypothesizes that adaptation to such changes would be faster and less error-prone if the tools also focused on automating translation from semantics to syntax, in addition to simply executing the syntax. With this approach, we build following low-maintenance security tools that prevent unauthorized sensitive data transfer at various vantage points in the World Wide Web ecosystem. We show how the security tools can take advantage of inherent properties of the sensitive information in each case, making the translation process automatic and faster:

- Appu, a tool that automatically finds personal information (semantics) spread across web services, and suggests actions (syntax) to minimize data loss risks.

- Newton, a tool that formalizes the access control model using web cookies. Using this formal approach, it improves the security of the existing session management techniques by detecting (semantics) and protecting (syntax) privileged cookies without requiring input from the site administrator.

- SilverLine, a system for cloud-based web services that automatically derives data exfiltration rules (syntax) from the information about sensitive database tables & inter-table relationships (semantics). Then, it executes these rules using information flow control mechanism.

CHAPTER I

INTRODUCTION

Many users of the Internet have at least a few accounts with online services. Users regularly trust such services with their personal information. Sometimes, this information is private, such as the person's name, address, or phone number, or family pictures. At other times, it includes highly confidential information like social security numbers, credit card numbers, and documents describing intellectual ideas. The amount of personal information submitted on the Internet is constantly on the rise. One statistic from the popular online social network, Facebook, which has 1.3 billion active users [19], estimates that on an average, a user uploads 90 personally generated content bits [88] per month. It should be noted that this statistic about personal content that has been uploaded by users is regarding just one service. In the future, this trend of sharing personal information with online services will only accelerate due to the following three factors:

- Reduction in data transfer prices: Due to efficient access channels (*e.g.*, Fiber Optics) [5], adoption of 3G/4G technologies [14] in regions lacking infrastructure, and increasing peering between autonomous systems, per unit data transfer cost is ever decreasing. If the uploading and downloading of documents is near instantaneous and cheap, then it would not matter if "Ctrl+S" saves the document locally or remotely.

- Wider adoption of cloud technology: More online services are using the cloud for computing and as a storage service [21] to build their own application logic, since it is cheaper, reduces management, and does not require one to reinvent the wheel.

- Personal convenience: When it comes to the trade-off between "complete control vs. continuous availability" of the information, more often than not, users choose the latter [21]. In addition to seamless access to the information from multiple places and devices, this also takes care of thorny issues such as backups and sharing.

Tangible evidence of this trend can be seen in the popularity of thin clients such as Chromebook [23] that exploit all three of the above factors.

1.1 Why Securing Personal Information is Hard

The unprecedented explosion of online information has also given rise to three main actors, with varying power and interests, to achieve the common goal of illicitly accessing this information. We briefly describe these entities, their vested interests and how they typically reach this goal. First, hackers often steal such information for personal financial gains [27]. Usually, this is accomplished by exploiting security flaws present in the application code (e.g. cross site scripting), third party frameworks, libraries, the database code, and even the operating system. Once a data breach has occurred, the hacker can increase the impact of the blow by trying to compromise the victim's other online accounts using already compromised information. Since many users reuse passwords, and usually the usernames are not confidential, the total brunt of the initial compromise is dependent on the victim's online behavior [29]. Once this data is stolen, the hacker has various ways of monetizing it, such as selling it to dark marketplaces [12], identity theft [9] or even blackmailing [2]. On some occasions, hackers might execute such thefts for seemingly altruistic reasons [10]. However, the end users suffer nonetheless from embarrassment, financial losses and sometimes with direct threats to their lives [8].

Second, since many of the online services are free, a commonly chosen way for a company to become profitable is advertising, giving rise to the $170 billion online

marketing industry [22]. Advertising networks are in a constant competition to display the most relevant advertisements to users visiting a general webpage to increase profit. For this targeted advertising, the ad networks try to gather as much information as possible about the visiting user, such as gender, age, ethnicity, location, past activities, and browsing behavior. Increasingly, this data gathering activity is becoming highly intrusive into a visitor's real life, since the ad networks can infer the real life identity of the visitor using objectionable techniques like record linkages, and from third-party leakages [84].

Finally, the most powerful entities in this set are state controlled actors such as the NSA, GCHQ, and The Great Firewall of China. These agencies often have control over critical junctions in the Internet infrastructure to tap and monitor traffic [11], and the resources to painstakingly put together not only current online activities, but also link the historically stored traffic traces. Such analysis is capable of creating a very detailed profile of the user [4]. Despite the presence of sound security measures such as cryptographical algorithms (RSA), and secure protocols (HTTPS), these agencies can often bypass them by straight strongarming [4] for mass surveillance, targeted monitoring with user deanonymization using similar techniques like ad networks, or just by exploiting presented opportunities (Heartbleed, Poodle). Even an effective anonymizing tool like Tor can become ineffective [60] against surveillance from such combined vectors along multiple axes.

Due to the rapid wallification of the web [85], services require users to create an account, and fill in profile information to use them. The presence of such multiple accounts containing user information combined with all the persistent threats mentioned above makes it likely that the user's personal information will be compromised at some point.

Existing guidelines, policies, and mechanisms to thwart these attacks are often ineffective due to the rigidity and inflexibility of such mechanisms. These mechanisms

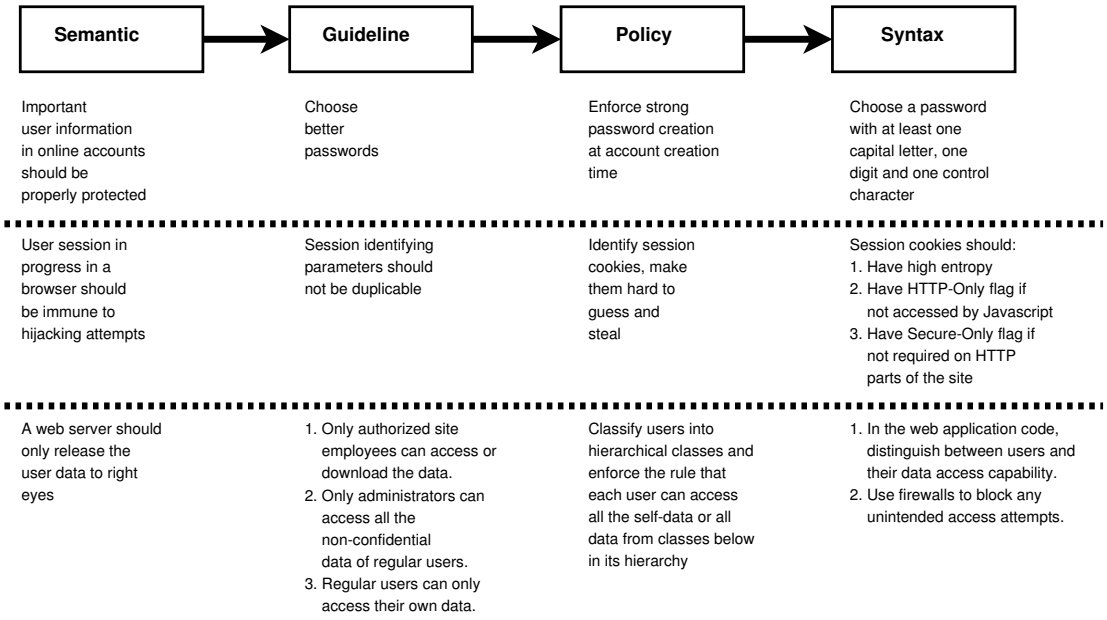


Figure 1: Security semantics getting translated into security syntax

are well researched and properly implemented. However, they are not sufficient to guarantee online safety.

1.2 Dissertation Contributions

Current security mechanisms focus on "how to stop illicit data transfer" (i.e., the "syntax"), and many tools achieve that goal in principle. Yet, the practice of securing data additionally depends on allowing administrators to clearly specify "why the data should be secured" (i.e., the "semantics"). Currently, translation from "security semantics" to "security syntax" is manual, time-consuming, and ad hoc. Even a slight oversight in the translation process could render the entire system insecure. Security semantics frequently need modifications due to changes in various external factors such as policy changes, user reclassification, and even code refactoring.

If we can reduce the manual intervention required in such a translation process, then we can adapt to changing threat models faster with minimal errors.

With this approach, we focus on building tools to thwart attacks mounted at three crucial vantage points in the World Wide Web: first, attacks where hackers

compromise weak passwords and steal the data; second, when the cookies present in the browser can be stolen resulting in session hijacking; and finally, when the web servers are compromised and data is transferred from the database en masse.

Figure 1 shows the existing solutions for these attacks with the current translation process from the semantics to the syntax. In each case, we design a security system that, in the face of constant changes in threat models and data access control, tries to reduce the gap between the policy and effective safety by automatically redrawing the syntactical security rules for changing semantics.

Password-based access control is ubiquitous and does not have a worthy contender. Unfortunately, this mechanism is often ineffective because of problems such as weak passwords, and password reuse. Suggested approaches to augment these mechanisms such as password managers or multi-factor authentication schemes are either not foolproof, and not always available, or not easy to use. Users choose their passwords when they open an account according to their perception of the importance of that account and their perceived use of it in the future. But this decision does not get reevaluated when this perception differs, making it static. We describe Appu in Chapter 1, a tool that spots discrepancies between user's mental model of required security versus the actual required security, and helps the user to rectify it. To do so, Appu automatically finds which data is important for the user, and the spread of this data intentionally (by the user from explicit sharing), as well as unintentionally (from leakage to third parties). We then suggest helpful hints based on derived model of site importance for a particular user suggesting how the account security can be improved.

Once the user has logged into an account, and the session is in progress, the access control is protected by the session identifiers or cookies. Thus, these cookies become security linchpins and if such cookies are stolen by an attacker, then the user session can be taken over, which is commonly known as a session hijacking attack.

Mechanisms to protect against this already exist, such as "HTTP-Only" and "Secure-Only" flags on cookies, HSTS, and HTTPS Everywhere. In practice, though, they often fail to work as intended due to various factors such as the reorganization of a site's structure. We describe Newton in Chapter 2 that automatically detects security issues in the cookie based access control mechanism. We discovered errors in cookie implementation in 113 sites of the Alexa top 200 sites. This includes high profile sites such as Yahoo, Amazon and Fidelity. We reported many of these errors to site administrators whenever possible. We got confirmation from 6 sites (Yahoo, Amazon, Vimeo, Magento, WooCommerce, and BigCommerce). In two of these cases (Vimeo and BigCommerce), they acted on our reports, fixing the error.

A user's data stored in the database at the server can be stolen by exploiting various bugs present in third party libraries, web application code, and database modules. Existing control mechanisms such as firewalls can be often bypassed, as such mechanisms are unaware of the privileges of each user session, and the importance of data present in each table. We describe SilverLine in Chapter 3 that automatically makes ex-filtration decisions based on each user's privileges versus privileges acquired by a user's session. To deduce this, SilverLine uses session cookies to identify session privileges and information flow control framework to track acquired privileges.

1.3 Roadmap

The rest of this dissertation is organized as follows. We show how a user can keep track of her personal data spread across web services in Chapter 3. We present the design of our Chrome browser extension Appu. Next, we describe Newton in Chapter 4, detailing how authentication cookies can be detected and protected. We describe how to protect data in web servers by tracking sensitive information flow using inter-table relationships and information flow control in Chapter 5. Finally, we summarize our contributions and discuss future directions in Chapter 6.

CHAPTER II

BACKGROUND AND RELATED WORK

The web has been under attacks of various kinds since its inception. In this chapter, we briefly take an overview of related work that covers such attacks and various proposed solutions that focus on the "security syntax." We show how the existing work defines the syntax very well, yet falls short to achieve the end goal of protecting the user data. We take an overview of three distinct yet related areas in web security and privacy. They are distinguished from each other because traditionally, the actors who execute the "security syntax" is different in each case. In the first section, we have solutions that attempt to improve user's online security and privacy where the syntax is mainly executed by the end-users of the World Wide Web. They could be assisted by the websites themselves by restricting their online behavior, sometimes in a user-friendly way. The second section defines the syntax for web application developers, laying the rules about how should they develop their applications. These rules are often standardized into guideline documents, and sometimes even baked into the popular frameworks to make the job of development easier. Finally, we take a look at diverse line of defensive tools available to website administrators to thwart data theft attempts in the presence of web application vulnerability. This dissertation proposes the technique of converting security semantics to syntax automatically. Using this technique, we build three security tools that improve the security status quo of their intended users.

2.1 Security & Privacy for Web Users

In this section, we offer a brief overview of existing work that focuses on improving users' online security. Web users often reuse passwords and that means if the hackers know about a user's password, multiple accounts of that user can be compromised. Attackers can usually get their hands on a user's passwords by compromising a specific web service and stealing the data for the entire userbase. However, if the user's passwords are one-way encrypted, it may take a while before the attacker can convert the passwords to plaintext. In the next section, we summarize corresponding research work which analyzes this problem from multiple perspectives, such as users' tendency to reuse passwords, tendency to choose sufficiently complex passwords that are hard to crack, users' mental models about how security works, and different attempts from researchers and websites to improve usability of security and privacy techniques.

2.1.1 Password Security Analysis

Narayanan et al. [96] showed that any memorable password can be cracked by carefully choosing the dictionary using Markov modeling. A study of password habits such as reuse and the chosen password strength of half a million users [63] showed that a user on average has only 6 different passwords that get reused across accounts. Followup studies since then have confirmed a high percentage of password reuse [81, 122].

Often users would take a base password, tweak it slightly across different accounts to avoid straightforward re-usability, and to make it immune to guess-ability. However, a recent study by Das et al. [54] shows that as many as 30% of such transformed passwords can be guessed with minimal effort after an attacker gets access to another transformed password from the same user. Given the frequency and the overwhelming volume of data breaches [28] an attacker can easily get a collection of passwords that were valid at some point.

Various mechanisms to estimate password strengths under different threat models

have met with varying adoption [41, 42, 79]. A recent study argues that solely relying on one single strength estimator results in poor estimation [129].

Despite problems with passwords, Bonneau et al. [43] found no other better web authentication mechanism that is more secure yet is as usable.

Various studies about [101, 126] password selection and reuse habits found that users internally have a mental model of important and not-so-important sites. They tend to use better passwords for important sites and also tend to less often reuse passwords used on important sites. In spite of this mental understanding that one should reuse important passwords less frequently, it was found that half of the users reused important passwords on less important sites [101]. Additionally, Haque et al. [67] found that frequently, it is possible to guess a user's password on important sites after obtaining a password on a less important site, even if the user is actively trying to differentiate between his accounts. Even if an attacker is able to correctly guess a user's password on another service based on an existing stolen password, he would not be able to exploit it without getting username. But Perito et al. [113] showed that usernames across sites are often predictable using Markov models. Additionally, it is possible to correlate usernames across sites using even less sophisticated techniques [73] as well as social engineering attacks [39] or deanonymization [24].

Security experts often agree that password managers are an effective way to deal with an overload of passwords. Even aside from other issues of password managers, such as single point of failure, recent research shows that password managers are not as secure as previously claimed [87, 123].

2.1.2 PII Leakage & Privacy Protection

An influential study by Krishnamurthy et al. [84] and other related studies [25, 26, 91, 92] have established, without a doubt, that personal information leakage to third party domains without the user's permission or control has become a norm rather

than an exception. Often, these leakages can become a tool in the hands of state actors to execute mass scale de-anonymization and surveillance where even a TOR like anonymizing system would be ultimately ineffective [60].

Recently, FORMLOCK [125] analyzed PI leakage to third party sites on contact form submissions. Since the majority of users would use their real names in such forms, this instantly de-anonymizes the user session.

Web privacy enhancing tools [7, 17, 18] provide varying amounts of prevention against such leakages. However, many of these studies and tools are either limited in scalability due to the amount of manual effort required to create accounts and detect the leakage, or are limited in effectiveness due to blacklist and whitelist rules. In contrast, we note that if a tool is aware of personal information about a user, it can enhance itself by creating such rules dynamically.

2.1.3 User Friendly Security

A recent study [78] about users' perceptions about their online data showed that, first, even the people who are technical experts do not take any significantly different action to improve their online security and privacy. Second, users often incorrectly think that they have not given a lot of personal information online, and are unaware about information accumulation over time. In spite of this unawareness, people are distinctly aware of the difference between anonymous and eponymous web browsing, and behave differently when they are under the impression that they are anonymous [40, 77, 111, 112]. An information exchange transaction is significantly affected depending on various permutations of anonymity for both sender and receiver [127]. But, as leakage studies above have shown, and perhaps due to the forgetfulness of users(*e.g.*, opening a Reddit account with an email containing a real name), often when a user thinks he is anonymous, and he behaves in a certain way, even when he might not be anonymous, after all. In such cases, an indicator of seeming anonymity vs. actual

identifiability would be useful to users.

Finally, some studies [49, 64] have confirmed that users can be persuaded with proper incentives to choose better passwords.

2.1.4 Information Flow Tracking & Control Systems

A huge research body exists in traditional information flow control systems. One of the major headaches for these systems is identifying what is sensitive information and hence should be protected. Recently, there have been some attempts to identify user's sensitive information automatically. AppIntent [137] attempts to predict whether a sensitive data transmission on Android devices is intended by the user by analyzing graphical interaction and symbolic execution. While this is an interesting problem, our work focuses on a different problem: identifying personal information spread on the web and its monitoring. Since we try to be generic and provide this detection mechanism generically, we cannot benefit from much more structured behavior of apps on mobile devices. Liu et al. [89] recently proposed a low deployment cost framework to detect personal information fields in Internet traffic at the network level. In addition to reliance on unencrypted traffic, the approach suffers from drawbacks such as it is hard to establish ground truth, and the participation is involuntary. Finally, a single user on the network might not learn anything useful about leakage concerning her data. ReCon [118] tries to provide some control and visibility over leaked personal information of mobile device users. They use machine learning techniques to automatically find PI at the network level. This work also has similar problems regarding establishing ground truth and encrypted communication. This growing research work to automatically find personal information, monitor it, and bring transparency, emphasizes the need for a tool like Appu. Once a user actually has an inventory of her footprint, other systems [1] can be used to keep track of the latest data breaches.

2.2 Web Authentication Cookies

In this section, we survey related work on web cookies. We survey related work in analyzing and computing auth-cookies for popular websites. There is extensive work in studying the use of auth cookies for user tracking (*e.g.*, [47, 119, 133]); our work is distinct from this body of work, as we focus on whether the cookies that are set by the primary domain for authentication are implemented correctly. We survey some of this related research below.

Identifying Auth-Cookies. A user's logged-in state to any part of a site can be formally represented as a disjunctive normal form (DNF) formula over cookies. Previous work focuses on identifying auth-cookies and checking if `HttpOnly` & `Secure` flags are set correctly. These approaches generally discover feature sets that distinguish auth-cookies from other cookies, train the cookie-identification algorithm using these features, and possibly attempt to guess the auth-cookies for a site based on this inference. The most closely related work to Newton is the contemporary work from from Calzavara *et al.* [48]. Using Selenium framework [20] the study found auth-cookies for popular sites by running tests in a controlled environment. Using this ground truth auth-cookies dataset, they identify distinguishing feature sets and train their algorithm to guess auth-cookies for unknown sites.

Newton, a security tool developed by us, is more significantly thorough than the algorithm from Calzavara *et al.* because it accounts for the fact that different parts of a same site can be controlled by different auth-cookie combinations and DNF equations. For example, the DNF for Google Search comprises auth-cookies `HSID, SSID, SID` but the DNF to access Google Mail requires `GX, LSID, APISID, SAPISID` and the DNF for Google Calendar requires auth-cookie `CAL` in addition to all the auth-cookies required by Google Mail. Calzavara *et al.*'s algorithm appears to be incomplete, as their study discovered only `HSID, SSID, SID` as auth-cookie

for Google, regardless of the portion of the site. The study also marked `APISID`, `SAPISID` as not being auth-cookie, even though Newton discovered that these cookies are in fact used for authentications on different parts of the Google site. The errors that Newton has uncovered in this previous analysis underscores the difficulty in computing the auth-cookies for each part of a website without performing a more exhaustive analysis. To allow for more exhaustive testing across a far wider diversity of sites and a larger user base, we developed Newton as a Chrome extension that can be deployed by anyone to compute auth-cookies in the field, which also allows it to crowd-source these measurements on a much larger scale than has been possible in any previous study.

Related work such as Sessionshield [100], Serene [56], CookiExt [44], and ZAN [128] attempt to find auth-cookies using features such as name, value length, and entropy, yet our analysis results demonstrate that none of these features are sufficient to completely identify auth-cookies. Serene [56] also protects the sites against session fixation attacks. Since successfully testing a site for this vulnerability usually requires two user sessions, one owned by the attacker and another owned by the victim, it is hard to automate this testing. Hence, we did not include testing for this attack in our tool.

Protecting Auth-Cookies. Newton also uses a different approach for protecting cookies in the user's browser. Whereas Calzavara *et al.* use the trained algorithm to guess auth-cookies and then possibly enforce the correct protection flags on them, doing so can miss important information and may also end up breaking the site functionality for a user. To understand this, suppose that a site has DNF: `(A & B & C) OR (A & B & D)` and that cookies `C` and `D` are protected with `HTTPOnly` and `Secure` flags. If their algorithm concludes that the auth-cookie DNF is `A & B` and enforces `HTTPOnly` and `Secure` flags, then that is unnecessary (since both terms in the DNF are already protected) and might end up breaking the site functionality if

cookies A & B need to be accessed by Javascript or over plaintext HTTP.

Developing Security Best Practices for Auth-Cookies. More than 13 years ago, Fu *et al.* examined the use of cookies for client authentication on the web and uncovered various vulnerabilities; since this time, of course, the web has become increasingly complex [65]. At that time, only single authentication cookies were used; although many of the recommendations from that work are now commonplace, the rise of complex web applications and authentication mechanisms begs for a re-appraisal of client authentication. The Open Web Application Security Project (OWASP) has set security guidelines that recognize the importance of this problem and also acknowledge that simply setting `secure` and `HttpOnly` flags on cookies is not a viable means of securing auth-cookies for many web applications [107]. However, even those guidelines are somewhat outdated and imprecise due to fast pace at which world wide web evolves. To the best of our knowledge, ours is the first work that makes these guidelines actually usable and provides a semi-automated tool to do so. In addition to this, some proposals [52] seek to replace the existing authentication cookies mechanism with one-time cookies which signs each request with the session key. While this idea is interesting, it would need adoption by all existing web servers and browsers in order to be effective.

2.3 Cloud-Based Security

We present related work in data isolation, information-flow control, and language-level taint tracking. Table 1 summarizes the features of many related systems.

Data Isolation. CLAMP [109] and Nemesis [53] isolate data flows from different clients. CLAMP isolates code and data for each user using separate virtual machines, each of which is given only a limited view of the database using a *query restrictor*. Thus, even if the Web server or the Web application is compromised, an attacker

Table 1: Comparison of related web security systems.

	RESIN	**Guardrails**	**DBTaint**	**Flume**	**Nemesis**	**CLAMP**	**SilverLine**
Can deploy across frameworks and languages	Requires porting	Only Ruby on Rails	Specific RDBMS	Yes	Specific framework	Yes	Yes section 5.4.1.3
Can deploy w/o changing application code	Yes	Requires Annotations	Requires DB schema change	Application needs to call IFC syscalls	Yes	Needs to change login and other modules of app	Needs to change login module 5.4.1.1
Performance	Limited due to instruction level tracking	No data available	Limited due to instruction level tracking	Reasonable overhead since process level taint tracking	Limited due to instruction level tracking	Limited due to slow VM spawning rate	Reasonable overhead due since process level taint tracking section 5.4 & 5.5
Data leakage prevention	✓	✓	×	✓	✓	✓	✓
Protection from XSS attacks	✓	✓	×	Limited	Limited	✓	×
Modularity	✓	×	×	×	✓	×	✓

can only retrieve data from the corresponding view. Due to the overhead of per-user virtual machines, CLAMP incurs higher overhead than SilverLine. CLAMP's query restrictor also restricts the Web application from performing aggregate queries (*e.g.*, COUNT(*), AVG). In addition, the query restrictor does not allow queries on sensitive tables if the user is not logged in. Web applications sometimes provide statistics, even for anonymous users, in which case an architecture like CLAMP would require Web application code inspection and modification. One such example is the "create users" command in OSCommerce. For even a moderately complex application like HotCRP [80] CLAMP's data access control policies become complicated and would require review from domain experts. CLAMP also requires creating a virtual machine for every user session, which creates scaling problems for large Web server applications. Nemesis separates security code from the application code by having a different user authentication and tracking system than the application code itself, and

authenticates the user twice, once at the application itself and then to Nemesis [53]. Nemesis places the Web application code in the trusted computing base, so it requires rewriting Web application code. CryptDB [114] protects user data by encrypting it and executing queries over an encrypted database to prevent unauthorized users from accessing data, but does not have any taint-tracking capabilities.

Information-flow Control. Information flow control has been applied to systems since the 1970s [57]. Many operating systems have secured a *single* host against exploits or prevented data exfiltration, starting as early as 1975 with the Hydra operating system [51] and continuing on to more recent work (*e.g.*, Taos [134]). *Decentralized* information flow control [94] was initially designed as a new programming language paradigm, and specialized operating systems such as Asbestos [131] and HiStar [140] have adopted this approach to allow OS-level information-flow control between trusted and untrusted processes. Dstar [141] extends HiStar's decentralized information flow control to the network, using signed messages to transfer labels and capabilities between hosts. Unfortunately, Dstar requires HiStar on both the client and the server to facilitate label transfers—installing a new operating system is often not practical for Web servers, and for our purposes it is unrealistic to require Web clients to run a particular operating system. Flume [86] modifies Linux to allow similar policies as SilverLine, but it still requires both trusted and untrusted applications be aware of information-flow control. In contrast, SilverLine allows applications to operate without explicit knowledge of information-flow policies (hence avoiding expensive application rewrites).

Language-level Taint Tracking. RESIN [139], Guardrails [45], PHPAspis [108] and DBTaint [55] use information-flow tracking at the programming language level to prevent information leaks. To prevent XSS and even injection, RESIN [139] allows

programmers to check and sanitize their code: a programmer identifies the data-flow boundary and defines policy checks to be performed whenever data crosses that boundary. GuardRails [45] transforms annotated Ruby source code into secure code by generating similar checks automatically. Unfortunately, these systems require application code to be rewritten, and bring the core Web application code into the trusted computing base. PHPAspis improves on this trust model by only trusting parts of the code and performing taint tracking for only untrusted third-party plug-ins; because it does not monitor all application code, it can only defend against a limited range of attacks. DBTaint provides a cross-application fine-grained taint tracking platform by tainting database records, which is similar to SilverLine's approach of adding taints to database rows. However, DBtaint does not define specific security checks, and it also requires a modified language runtime.

Full-system Taint Tracking. Static "tainting" approaches [68, 94, 110, 130] can tag variables or portions of memory with security labels or bits, but these methods typically require rewriting applications to be aware of information-flow control. Dynamic taint analysis techniques such as TaintTracker [98], a mechanism to monitor information flows at the instruction level to detect potential exploits on a host; and process coloring [75], which tracks interactions between resources ("color diffusion") used for early detection of resources on a host that possess "colors" of a vulnerable process. Other efforts include Panorama [138], Privacy Oracle [76], TaintDroid [59], and Neon [142]. Although these systems can track data leaks from legacy applications, they (1) track information flow using a fixed set of non-expressive taints, (2) focus on data leaks from a single host, and (3) apply static policies only (*e.g.*, tainting all data from external sensors). These systems also require instruction-level instrumentation, which introduces more overhead than SilverLine.

CHAPTER III

MONITORING SENSITIVE INFORMATION SPREAD IN THE BROWSER

3.1 Introduction

In this chapter, we focus on security tools that are designed to minimize the data loss risks originating from users' habits of sharing personal information online, and insufficiently protecting this information. The preventive security mechanisms to minimize these risks in some cases could have well-defined "security syntax" e.g. use a password with at least one capital letter and a number. Eventhough this specific syntax is widely practiced by the web services, it is not necessarily accurate. For other cases, such as actions to be taken to reduce further impact of a compromised password, the syntax is fuzzy. When the syntax is to be executed by the end-user, just showing them abstracted generalized reasoning does not motivate the user to actually act on it. Instead, if the tool provides them with intended benefits of executing that rule for their specific case ("semantics") then it is more likely that the users will act on it.

Loss of personal information from online accounts either through openly illegal activities [25, 26, 28] has been a point of endless discussion for policy makers, news outlets, the security community, privacy practitioners, and of course, end-users. In the past few years, there have been advances in the privacy community identifying personal information leakages in the Web [60, 84, 91]. The security world has no shortage of research work analyzing the common web authentication mechanism,

namely password based access control, that ranges from statistical analysis [41, 42], to highly complex mathematical models [96], to even behavioral [40, 77, 111, 112] and psychological [49, 64] analysis of users.

Despite these efforts, the end users' plight has not improved. We think that the main reason is a lack of joint efforts between the two communities, with the end user as the center of focus. The privacy community focuses on tools [6, 7, 17, 18] that are too impersonal for normal users (*e.g.*, tools that create blocking rules based on laboratory experiments). These tools are fragile since the coarse grained rules are created in a controlled setting, rather than adopting dynamically as the user interacts with them. On the other hand, the security community has over analyzed and created such a myriad of options and checklists [50] that users usually tune out and ignore the too detailed policies completely. Some of the recent work in privacy [83] has also identified this gap and the focus of privacy tools on syntax rather than semantics.

Past research has indicated that user habits can be changed via reasoned persuasions and incentives [49, 64]. We argue that such if the user is provided with a logical yet personal argument, then they can be won over to take the right action. Such a personal reason can only be provided by a tool that familiarizes itself with what information matters to each individual, and customizes its advice.

We have designed a browser extension, Appu, that tries to connect this gap by monitoring users' online activity. Appu automatically connects the online persona of the user with her real-life identity and warns the user when the two start to dangerously intersect.

The contributions of this work are as follows: First, we automatically detect the user's sensitive data by actively accessing their online accounts. Once this ground truth about real life identity of the user is put together automatically, we can monitor the voluntary spread and surreptitious leakage of this sensitive data across sites, as well as detect if it is sufficiently protected. This process is shown in Figure 2.

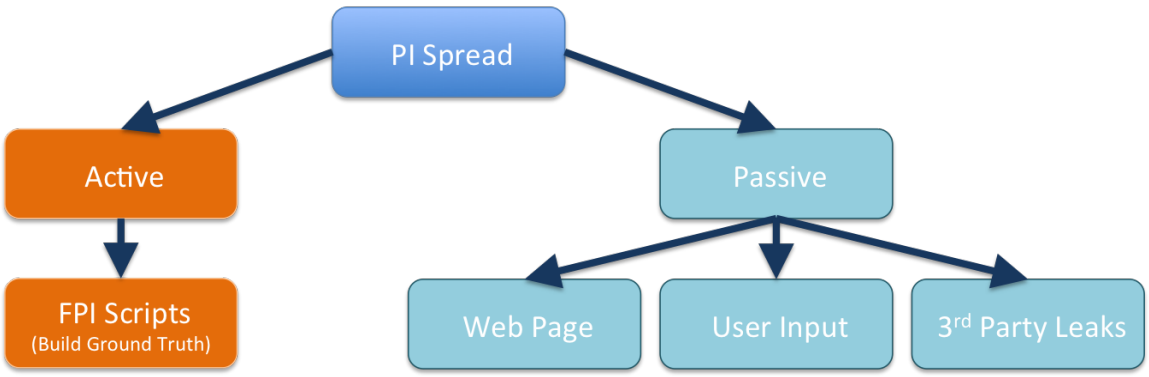


Figure 2: Taxonomy of Appu's monitoring actions

This provides the user visibility and some control over the user's personal data footprint. Second, we devised a mechanism to objectively decide site importance for the user. Using this mechanism, Appu creates a tailored classification of user accounts by deducing a user's implicit trust in such sites, implicitly expressed via shared data. This relieves the user from the burden of ad hoc classification of the accounts, which is often subjective and mostly wrong due to leaky abstractions invariably present in such mental models. Third, based on the collected PI and our classification system, we give users gentle nudges to choose better passwords on accounts that are insufficiently protected. Based on prior research, we hypothesize that users would respond to a reasonable and logical arguments such as "Since your PayPal contains social security number, choose a better password," more positively rather than impersonal warning prompting to just do so, which the user would often train herself to ignore over time. Finally, we also monitor for PI leakage to third parties. Our system is not based on rigid white or black list rules. It is capable of deriving these rules by monitoring the PI information spread, and hence is more flexible. Additionally, if the user chooses so, these leakages of PI are reported anonymously to our central server, which upon getting enough confirmations, can publish a public advisory list.

3.2 Threat Model

Our threat model focuses on mistakes occurring due to not following standard security rules, and also due to lack of visibility about what is going on with users' data.

Accounts secured with wrong passwords. The straightforward case of having the wrong password is the one that is weak (easily susceptible to dictionary or targeted personal attacks). When such a weak password is reused, a compromise at one site can jeopardize rest. Secondly, passwords could be derived from a base password, but the pattern is guessable, rendering derived passwords too similar. Finally, passwords could be strong, but could have already been leaked from previous hacks. Thus, if the server with that password gets compromised, and the attacker acquires an encrypted version of the password, brute-force cracking would be easier, as that password would be present in the attacker's dictionary. Password reuse with different usernames only provides a false sense of security, since usernames could be guessed.

Third-party leakages. We also want to monitor (and possibly thwart) PI leakage to third-party domains. Existing tools mostly function based on black or white listing rule sets. However, an ideal system should be adaptable based on a user's data in the field, and also show her which exactly which personal information has leaked and if there is cause for concern.

3.3 Dynamic Classification of Sites into Importance Categories

A site's importance must be decided based on a user's implicit trust on the site exhibited by what kind of data she feels comfortable sharing with the site[1]. As

[1]We acknowledge that a more better way of classifying the site should also consider site's reputation and conformation to security standards. However, due to the lack of well established repository about such knowledge, we defer to user's discretion.

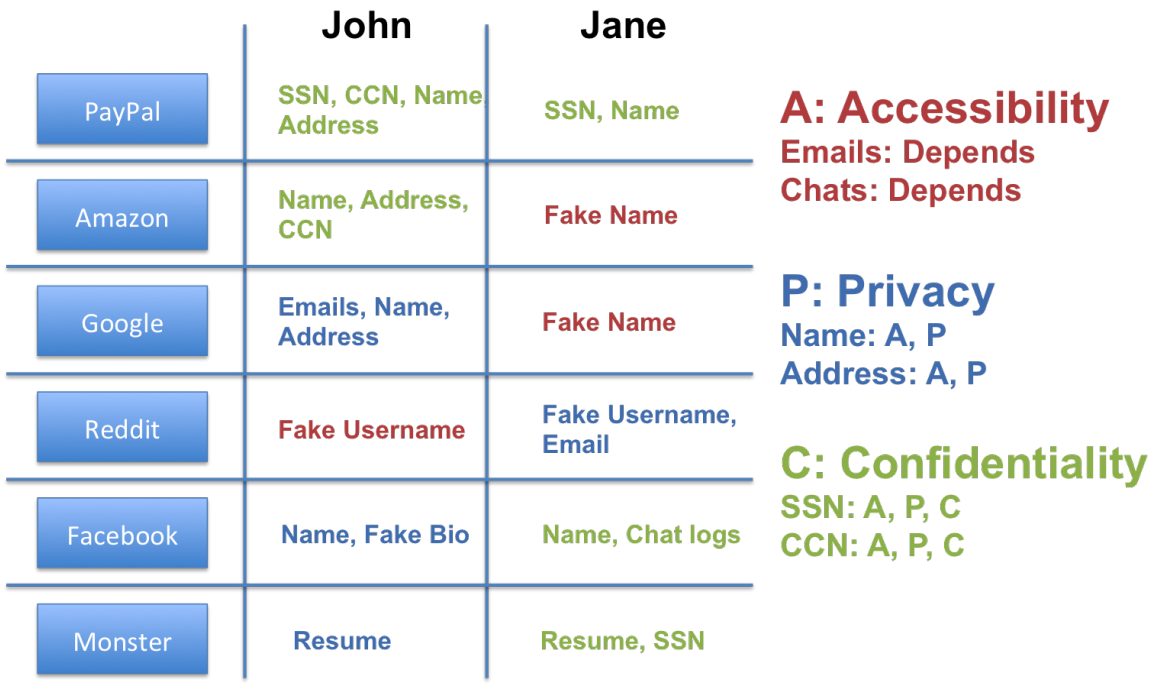


Figure 3: Two users have accounts on same websites. However, based on the information they have shared with each site, the sites classification into importance categories can be totally different. Green color indicates confidential, blue indicates private, and red indicates trivial information.

shown in Fig-3, let's say two users have accounts on PayPal, Google, Amazon, Facebook, Reddit, and Monster. An intuitive classification of these sites into security classes based on their services, number of users, and usual interactions with such sites, would result in three classes: MOST-SECURE{PayPal, Google}, MODERATELY-SECURE{Facebook, Amazon, Monster}, and LEAST-SECURE{Reddit.}

Suppose that each user has shared certain bits of personal information on each account. In this case, site importance for each user is distinct. To see how, we define three attributes desired for any password, and when they should be set, as follows:

- Accessibility: At the most basic level, each password has this attribute set since the main use of password (with username) is to identify yourself to the site and access your account.

- Privacy: When a user shares information that can identify him in real life, the

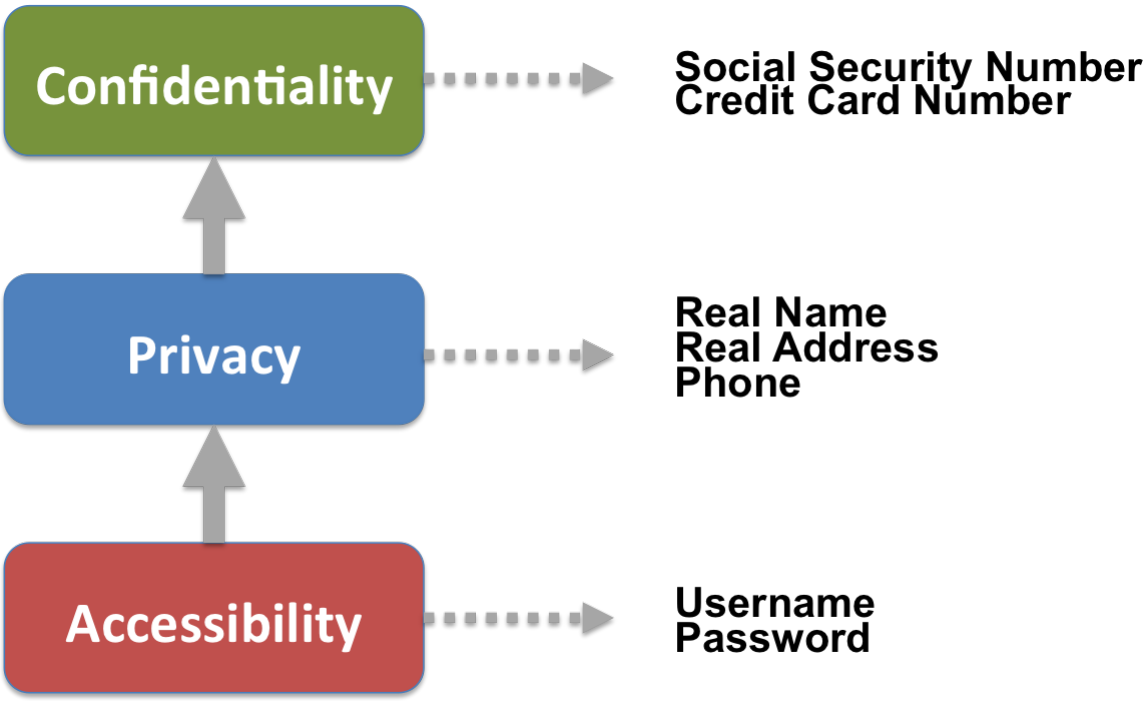


Figure 4: Hierarchy of accounts based on type of information contained. Loss of confidential information causes financial distress to the user. Loss of private information can deanonymize user.

password acquires privacy bit. If such a password is compromised, the user will lose his sensitive information, which may not cause any distress (at least until combined with other information).

- Confidentiality: When a user shares information such as a social security number which is strictly confidential, the password acquires confidential attribute. Most sites do not store complete credit card numbers. Seemingly, it is pointless to make a password acquire a confidential attribute upon sharing it. But, if such an account is compromised, the attacker may simply change the address and place orders.

With this, we see that for both users, PayPal's password acquires all three attributes. However, since Jane has account on Amazon but has never ordered anything, it only

acquires "P" attribute. Since John has a lot of email exchange containing many confidential documents on Google, its password acquires all three attributes. On Facebook, John only has given her real name, giving that account a privacy "P" attribute. Jane spends a lot of time on Facebook and her account contains lots of chat session with real life friends detailing personal, and private stories, thus her Facebook password acquires all three attributes. At first glance, for both users Reddit's password may seem to acquire only "A" attribute. However, when Jane first opened the account, she provided an email with her real first and last name. So Reddit's password for Jane acquires both "A" and "P." Finally, on Monster, both have submitted resumes (hence "P" attributes), but Jane also has given her social security number, hence also a "C."

With this, the classification for each user is:

- John: MOST-SECURE{PayPal, Amazon, Google},
  MODERATELY-SECURE{Facebook, Monster},
  LEAST-SECURE{Reddit}

- Jane: MOST-SECURE{PayPal, Monster, Facebook},
  MODERATELY-SECURE{Amazon, Google, Reddit},
  LEAST-SECURE{}

Passwords can be reused (although this is not advised, at least in the case of MOST-SECURE) in accounts that belong to the same class. The above classification for John and Jane may ostensibly seem straightforward and nothing of novelty. However, one must realize that these pieces of personal information are acquired over time. Thus, Jane may over time forget that she has shared her email address containing her real name and keep using weak passwords. This could result in account compromise (by targeted or mass compromise events), associating Jane's real life identity with comments she shared on Reddit under the trust that she was anonymous. Additionally, one should also keep in mind that such implicit trust relationships between a user

and her account are subject to change. When a new service is introduced, she may open an account just to give it a try, and in that case it is okay to choose an easy and memorable password. However, over time, she may start using it much more because the majority of her friends are also on it, and also start trusting it with more data. Similarly, there have been plenty of examples where mass exodus from a service has occurred [3, 16]. In such cases, what happens to data shared earlier on that account, and also if that password was reused in other sites in that class, how should this trust downgrading affect those sites?

Often, users fail to capture these changing trust relationships between accounts, as they themselves are not consciously aware of the changes.

Finally, an ideal solution from the security world such as always using complex distinct password for all sites and keeping changing passwords frequently fail in the real world. Similarly, many users are wary of using password managers for reasons stemming from feeling uncomfortable with trusting somebody managing such data for you [15]. It also creates many questions in the event of management change and could make users doubtful [13]. Finally, as current research shows, password managers themselves are not as secure [87, 123].

3.4 Design

To capture the implicit "trust-relationships" between sites and a user, Appu must first automatically detect the user's personal information with minimal input from the user. Once this information is detected, it must be monitored to see how it is spread explicitly by the user (indicating change of trust level) and unintentionally to third-party domains (indicating leakage of such information). Also, a user would have already had some footprint before Appu is installed. Estimating such past spread is challenging. Additionally, it is challenging to detect password reuse, password similarity, and already cracked passwords without compromising or revealing the current

Table 2: Personal information fields that are currently monitored by Appu

Actively Populated	Passively Monitored
name, email, phone, ccn (last-4), ssn (last-2), address, occupation, employment, relationship-status, country, city, zipcode, hometown, anniversary-date, birth-date, gender, known-languages, religious-view, political-view, graduate-school, high-school, interested-in, first-name, last-name, middle-name, ethnicity, income, education, height, eyes, hair, have-kids,	password-strength, password-length, password-reuse, password-similarity, password-leak-status, username, username-length time spent on site (logged-in as a particular username), time spent on site (total), entered input length, metadata about uploaded files (size, type, name),

password. Finally, once Appu detects this spread, the question is how to inform the user, as well as nudge her to take corrective action. Each of the following subsection tackles these challenges.

3.4.1 Detecting Personal Information of the User

Table 2 shows personal information fields that are monitored by Appu, and the way each field is populated. We first start by describing how Appu automatically populates confidential information about a user, such as real life name, address, phone, age, gender, birth-date, etc.

For this, we note that usually financial institutions vet such information by some external methods such as verification, background checks, or government issued identification cards. Upon such verification, this information would also be present on the user's corresponding online account. For example, on Amazon, when one visits payment management options, one can see all credit cards associated with that account. These credit cards also contain user's real name, address and phone numbers. Thus, if we could somehow access this verified information for each PI field, then we could accurately construct the user's real life identity, protect spread, and monitor PI leakage. Figure 5 shows an example of a hypothetical populated personal store after

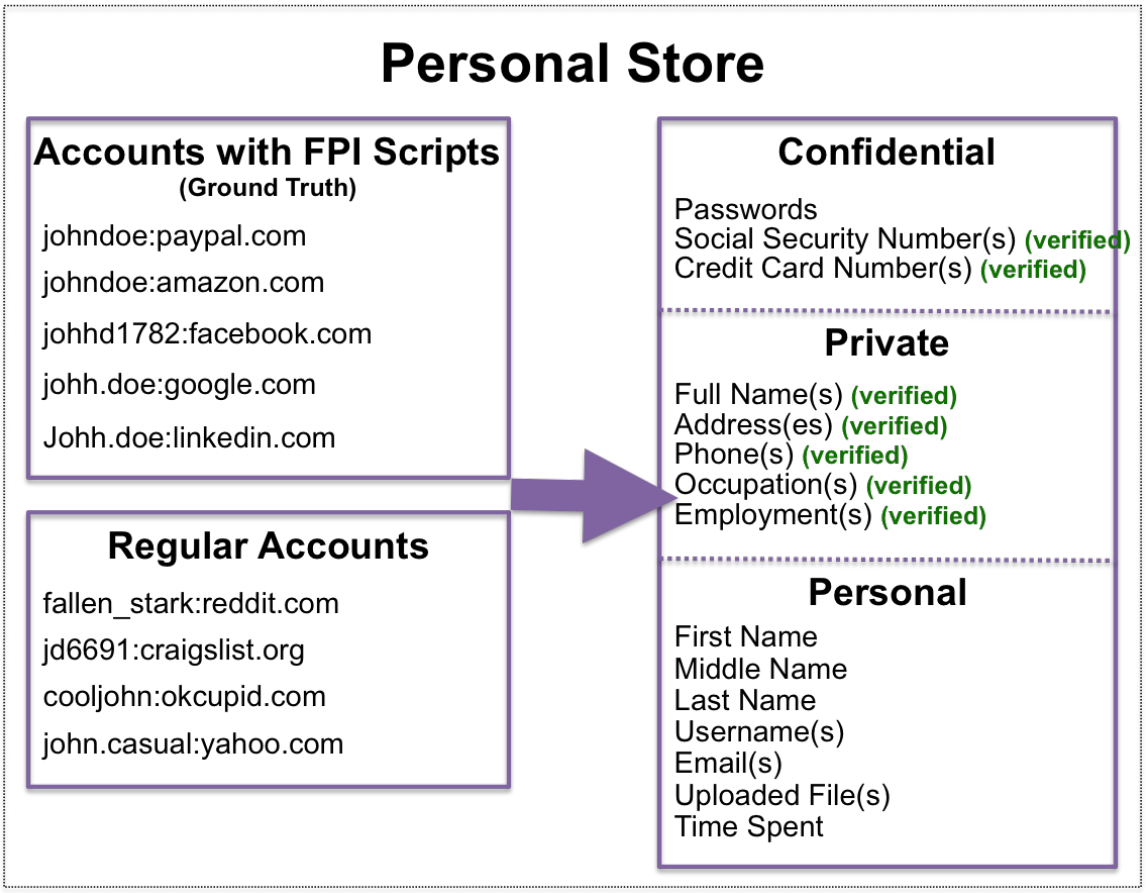


Figure 5: An example of how populated personal information store may look like after initial FPIs have executed. Left side shows various accounts for a user and right side shows populated personal information fields.

Appu has automatically detected information.

3.4.1.1 Fetch Personal Info (FPI)

To access this information, Appu first detects login events on any of the sites where such vetted information could be present. Once the user enters username and password, and logs in to the account, all we have to do is get the user's permission to access this information, and navigate through her account to locate and store necessary information corresponding to the username with which user logged in. Unfortunately, there is no easy and reliable way to locate and download this information. For this reason, we created our own mini-language called as "Fetch Personal Info" or FPI. FPI

Table 3: Main commands of our information scraping mini-language Fetch-Personal-Info (FPI). Each of these commands can additionally be tweaked by several parameters.

FPI Action	**Function**
`fetch-url`	Fetch a hardcoded static URL
`fetch-href`	Locate the HTML element with selector, & fetch dynamic URL using element's href attribute
`fetch-dom-element`	Reduces the scope of further FPI actions to children of fetched element
`simulate-click`	Simulate a mouse click on a specific element on the page. Wait for click success using provided confirmation condition
`store`	Store the selected value with given key (*e.g.*, Name) and attributes (*e.g.*, Verified)
`combine-n-store`	Same as above but used for logically single values split across multiple elements

language is written in XML.

FPI language consists of six commands to navigate and download user information. These commands, along with a brief description of each, are shown in Table-3. Each of these commands can be further modulated by passing various parameters as attributes. Each node consists of an action which is one of these six commands, a CSS selector and any number of children nodes. When the FPI interpreter executes these commands, it first applies the CSS selector on HTML elements passed to it by its parent. Then it executes the command on the result. After that, it passes the result of executing the command to its children nodes. Children operate on their own CSS selector and action command on this result. Thus, the root node operates on the entire HTML document, but each child's selectors and actions are applied to a smaller scope. However, all of the sibling nodes in FPI operate on the same set of

HTML elements generated by their parent. Most of the commands from the table are self explanatory. We will just briefly mention the "simulate-click" command. It is required because often some data would only be revealed when the user actually clicks on a specific region of the webpage. Internally, the webpage may just reveal already present information, or make an AJAX call in the background to fetch that information. For example, when a user clicks on his Facebook account name, she is shown the "first", "middle", and "last" components of her name. In the background, an AJAX call is made to fetch this information.

With these action commands, we could navigate all of the PI information in the 70 most popular site accounts such as Facebook, Google, Amazon, Live, Yahoo, etc., as well as financial sites such as PayPal. We note that with some understanding about CSS and Javascript, one can write an FPI script for any site in less than 15 minutes on average.

Figure 6 shows an example of partial FPI written for Amazon.

3.4.1.2 How To Write an FPI Script for New Account

To write an FPI script for a new site, first one has to decide which personal information needs to be downloaded from that account, and whether it is verified. After that, one manually locates this information. Writing just an FPI to get this information is easy with some knowledge of CSS selectors. Although one faces the following challenges to write a good FPI script that is immune to page structure changes.

FPI Scripts Challenges A hastily written FPI would break easily if the page structure changes even slightly. For example, suppose the information of interest is present as a value in 12th `<DIV>` leaf node. If this `<DIV>` does not have an ID attribute (which is frequently the case) and the selection by classname results in multiple `<DIV>` elements, then one would have to do some CSS magic such as "N-TH" child which in this case would be "div:nth-child(12)." However, such an FPI would be

```
<div name="amazon">
  <action type="fetch—url">https://www.amazon.com</action>
  <div name="your account">
    <action type="fetch—href">
      :contains("Your Account"):not(:has(*))a:first
    </action>
    <div name="account setting">
      <action type="fetch—href">
        a:contains("Change Account Settings")
      </action>
      <div name="name—div">
        <action type="fetch—dom—element" jquery_filter="ancestor—0">
          :contains('Name:'):not(:has(*))
        </action>
        <div name="name" can_be_a_null="no">
          <action type="store">
            span.ap_cnep_val
          </action>
        </div>
      </div>
    </div>
  </div>
</div>
```

Figure 6: Example of an FPI partial script. This script will navigate a user's Amazon account after logging in and download his name. Action commands in blue are navigation directives whereas rest of the actions are for scraping the information

very fragile and immediately break when another <DIV> leaf node is added anywhere above our value of interest. Considering the feverishly fast web development pace, this would result in FPIs that break and need to be fixed often.

Additionally, as shown in Figure 7, one also faces challenges such as widely different document structures for the same information representation. To store single logical values split across multiple nodes shown in Figure 7, we designed command "combine-n-store."

Writing Stable FPIs By Exploiting Information Content To solve challenges in writing sturdy FPI script, we exploit the information representation along with document structure as shown in Figure 8. We note that usually information presented

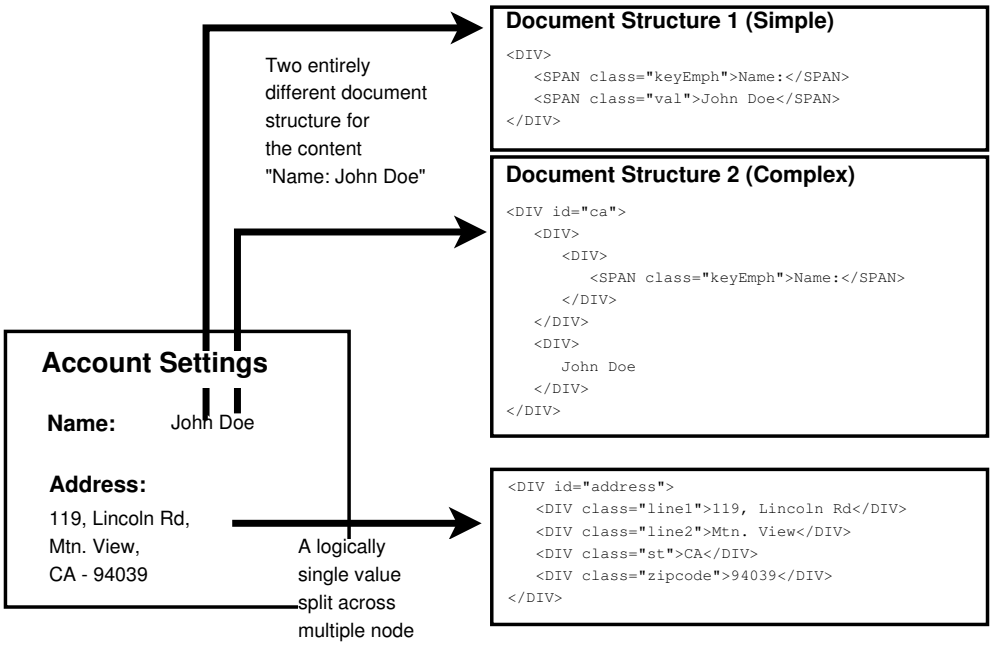


Figure 7: Difference between information presentation and structuring in web pages

to the user is "KEY-VALUE" pair, with the actual "KEY" such as "Name" or "Address" rarely changing even if the underlying document structure may changes many times. With this trick, to locate any information, we first locate its key, which is used as an anchor in the document. Once we locate the key, we traverse up the document tree to the common ancestor, and finally a command to walk down from common ancestor to leaf node containing the actual value. Thus, no matter how rest of the document structure changes, as long as the subtree from `<DIV id="ca">` is unchanged, the FPI continues to work. From our observation, usually, such small subtrees rarely change.

Value Normalization Often, different fields in personal information store contain widely different values. For example, a phone could be represented by formats such as "yyy-yyy-yyyy" as well as "(yyy) yyy-yyyy." Date representation also varies widely. One of the most challenging values to be sanitized is addresses. A user might enter the same address as "Lincoln rd, NW" and at other times as "Lincoln road, northwest." Appu needs to detect that both of them are same addresses. We used Google maps

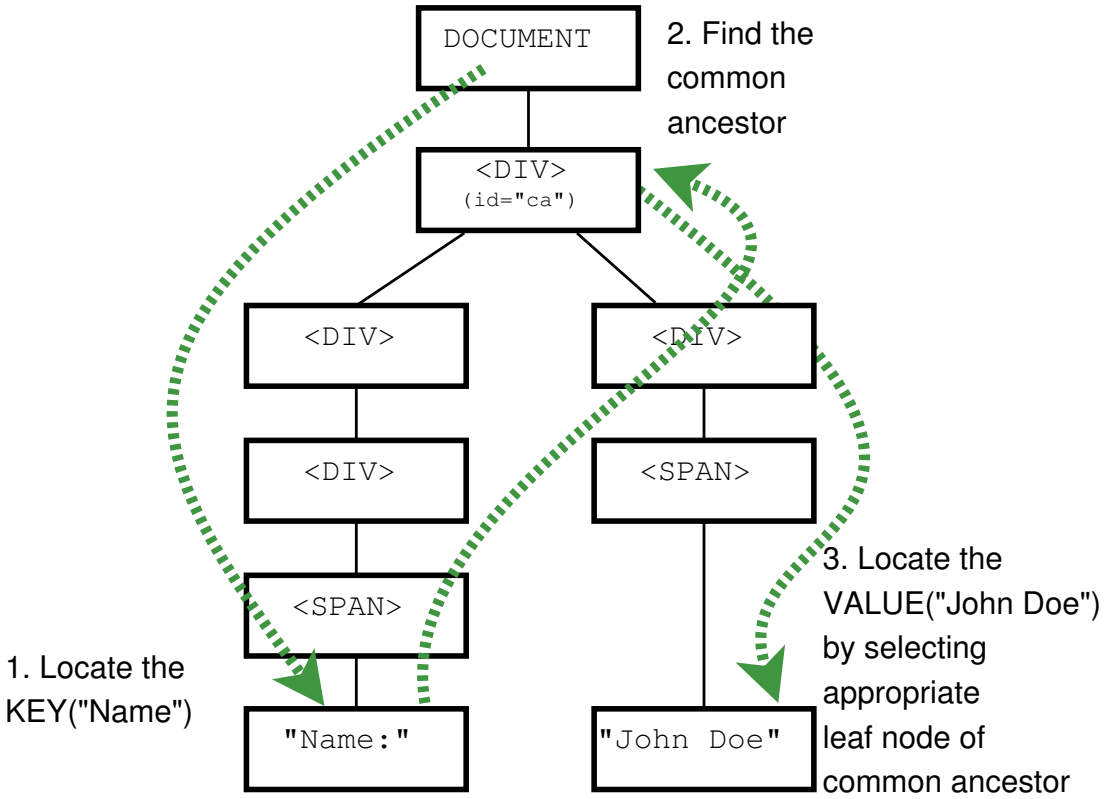


Figure 8: Locating desired information by exploiting both information representation and document structure. Green arrows indicate the located information in each step. Each box represents an HTML tag.

library to convert any address in the USA to canonical format of street number, street name, city, zipcode, and country. Finally, for confidential values, we only store the last two digits.

Reliability of FPI Scripts In spite of writing a sturdy FPI script, a webpage may change and the script may break. In this case, the FPI would either stop working or collect incorrect data. Omission failures are easily detected and reported by the FPI interpreter to the Appu's central server. Usually, commission failures are harder to detect. But, in Appu they tend to get detected because of two reasons. First, all FPI scripts are periodically executed to monitor changes to account profiles. Second, due to passive monitoring of webpages for PI, if Appu detects presence of PI that was successfully downloaded using the FPI script before the script started working

erroneously, a value mismatch would get reported.

Since we first created the FPI scripts for major accounts like Facebook, PayPal, Google, and Amazon, all the initial FPI scripts for these accounts were broken due to website or webpage reorganization. Fixing these broken FPIs took less than 10 minutes in each case.

3.4.2 Monitoring the Personal Information Spread

A personal information store in Appu consists of all the PII belonging to the user that is downloaded with the help of the FPI scripts. Once the personal information store is populated by the initial execution of FPIs, Appu starts monitoring the presence of populated PI passively.

3.4.2.1 Detection of Past Spread

Every user that starts using Appu already has some footprint spread on the web. In fact, initial seeding of FPIs work exactly for this reason. However, due to this, estimating exact PI spread becomes hard. To solve this, whenever a page gets loaded, Appu checks for already downloaded PI, and notes its presence on sites for which we have not written any FPIs. This estimation depends solely on the assumption that the user would end up visiting a webpage such as "profile settings" by chance. Thus, this is not a completely reliable method, but over a period of time, it would give a fairly good estimate. Once again, detecting a PI value on a webpage is hard due the varying formats of each personal field type. We once again exploit the fact that most of the PI values have a key that can be used as an anchor to detect the value. After that, we use each personal field type specific detection and normalizing mechanism. For example, in the case of addresses, we find each major element of the address such as "street number, street name, zipcode," or "street number, street name, city, state." If the radius of the smallest circle containing all of these elements is less than an experimentally determined threshold, we count that as a match. Even so, our

detection algorithm is not fully correct, but we are working on techniques to make it better. All other common sense tricks apply in these cases such as if the PI field type's value entropy is low (*e.g.*, "Gender"), then usually do not attempt to detect such values to avoid false positives. In the future, we plan to consider the category of the site for attempting to detect certain PI types. For example, on a dating site or social networking site, one is highly likely to find the "Gender" PI type rather than on an eCommerce site.

3.4.2.2 Monitoring Changing Trust Level

In addition to monitoring webpage content for PIs passively, Appu also monitors for PIs in the text entered by the user. Once again all challenges and tricks used to detect PI present on webpages applies here. If we detect a match, we add that site to that PI value. Eventually, if a user shares many more PIs causing the site to change its security class, we would prompt the user to adjust the discrepancy. This method obviously has some limitations, such as we do not do sensitivity analysis of the user's text content using something like natural language processing. Similarly, when a user uploads files on sites, we do not parse their contents for sensitive information. In such cases, we just measure the amount and type of information being uploaded by the user. For example, the number of characters typed in by the user. Also, if the user uploads multiple document types on Google, but only images on Facebook, then it is more likely that she would trusts Google with more confidential data. We are aware that detecting signal from such a high level, eagle's eye view might not necessarily portray an accurate picture. In future versions, we plan to implement a better strategy to deal with it.

Usually, during passive monitoring, we try to match against already known PI data with the exception of username and passwords. Whenever we detect a login event, we add the username to PI store (while also calculating its closeness to the

real name of the user).

3.4.2.3 Dealing with Passwords

User passwords should be always handled very carefully. In this section, we mention how we calculate each statistic on user's passwords.

Password Strength We use open source password strength estimation library `zxcvbn` [30]. It estimates the strength of the password as information entropy bits. This is a lightweight library with only essential dictionary values on various types such as the most popular top 10,000 names. Thus, the strengths estimated by this library are lower bounds, meaning if it estimates a low strength then certainly the password strength is bad. However, a high estimation does not necessarily mean that the strength is higher. Since Appu runs in the user's browser and we do not want to violate the user's trust by sending her password outside in any form, we have to compromise on a lightweight library. Even then, we see that it estimates low strengths for a shockingly high percentage of passwords in actual use; many of which are still being used to protect high security sites such as financial or banking sites.

Password Reuse To detect reuse, one must compare passwords. Since we cannot store passwords in plain text, we first add user specific salt and then one-way encrypt the passwords a million times. Currently, we use sha512 to one-way encrpyt the passwords. However, a better choice is bcrypt which would be included in future versions.

Password Similarity Once again, due to the self-imposed restriction on password storage method, we cannot directly detect similarity between two passwords. We use MinHash algorithm to estimate similarity. However, hashes of N-grams can be used to detect the actual password if the attacker gets them, so we keep these N-gram

hashes in the memory for a limited amount of time such as 15 minutes. If the user enters another password within that time, it will estimate the similarity.

Password's Leak Status There is a large corpus of passwords available from all the data breaches that have occurred over the last few years. Most password crackers have already incorporated this large corpus of passwords in their dictionaries for brute-force password cracking. If the user uses one of such passwords, and if the site where he is using it gets compromised, then even if the password is properly stored by the site, an attacker would detect the original password with high probability. To check if the user is using one of such passwords, it needs to be compared against these known cracked passwords. However, the total corpus of such cracked passwords in our possession from previous cracks is about 45 million. Appu cannot download this to the user's machines as its size is in few GBs. To detect if a user's password is present in this list, we use bloomfilter at the server. The bloomfilter itself is 50GB long. Appu calculates bloomfilter bits and requests the server to check for any matches. If the match is found, then that password is marked as insecure. One problem with this scheme is that an "evil server" offering such a checking service, can keep track of each plain text password and its corresponding bits. When Appu reports that a password for a certain account is cracked, the "evil server" knows the password and account exactly. However, since we are running this service, it is a fair justification that we do not try to correlate and find user's passwords. In the future, this feature would be "opt-in."

3.4.2.4 Dealing with Other Authentication Mechanisms

In the current version of Appu, we do not detect OpenID based logins. We also do not detect if a user has enabled two factor authentication. We intend to tackle that in the future version. However, in both cases, once the user has logged in, if there is known PI present, it will be located.

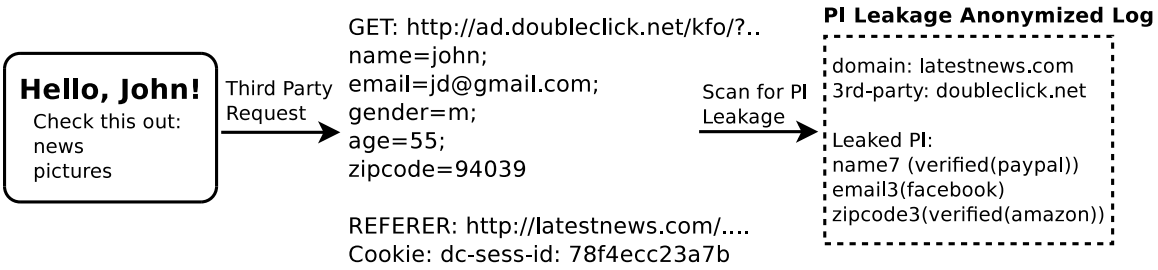


Figure 9: Monitoring and reprting third party personal information leakages in Appu

3.4.2.5 Monitoring PI Leakage to Third-Party Domains

Finally, for every HTTP request originating from a browser tab, we compare the original domain with the requested domain and if there is a mismatch, we look for the presence of PI in the requests parameters and referer values. If the PI value being detected has low entropy in general, then a successful match is reported, but it is less significant to high entropy matches such as username, name, or email. Additionally, we also look for common key names such as "gender." Detected leakages can be anonymously reported to the central server as shown in Figure 9. As shown in the figure, instead of reporting actual PI value, a metadata such as whether it was email or username or gender would be reported. If the server gets confirmations for such leakage from multiple reports, then it would be confirmed. A published report in such cases would act as a privacy violations clearinghouse.

3.4.3 Spread Visibility & Corrective Actions

In this section we talk about how all the collected information is converted to actionable items by the user.

My Footprint. To increase visibility and a user's awareness about her footprint, Appu presents this information to the user as a compiled report as shown in Figure 10.

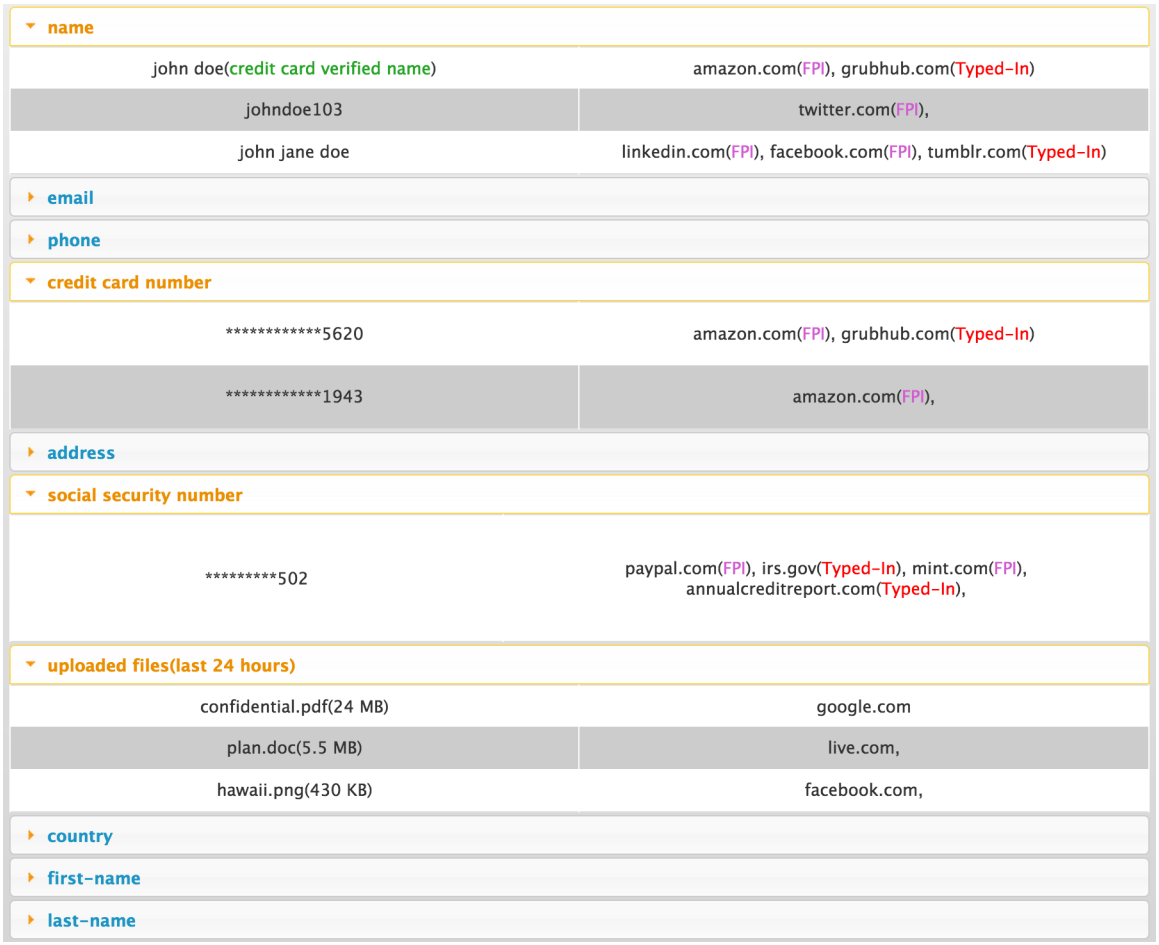
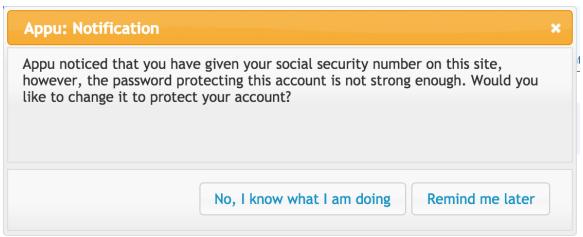
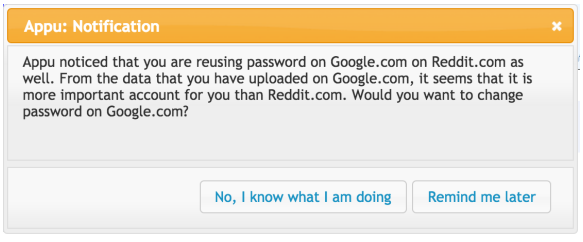


Figure 10: User can view the sensitive information monitored by Appu. Information is categorized according to whether it was actively downloaded (FPI), or passively captured (typed-in or uploaded files). Also, some information such as name associated with credit card information allows Appu to distinguish the real names of user. None of this information is sent out in reports.



(a) Appu detecting that a site's security class needs to be upgraded.



(b) Appu suggesting a class change for Google based on user's implicit trust expressed by sharing more information on Google.

Preemptive Password Change Suggestions. Figure 11a, and Figure 11b show Appu nudging the user towards more secure behavior with a logical argument tailored to that user's actions. We do not redirect the user to password change page since

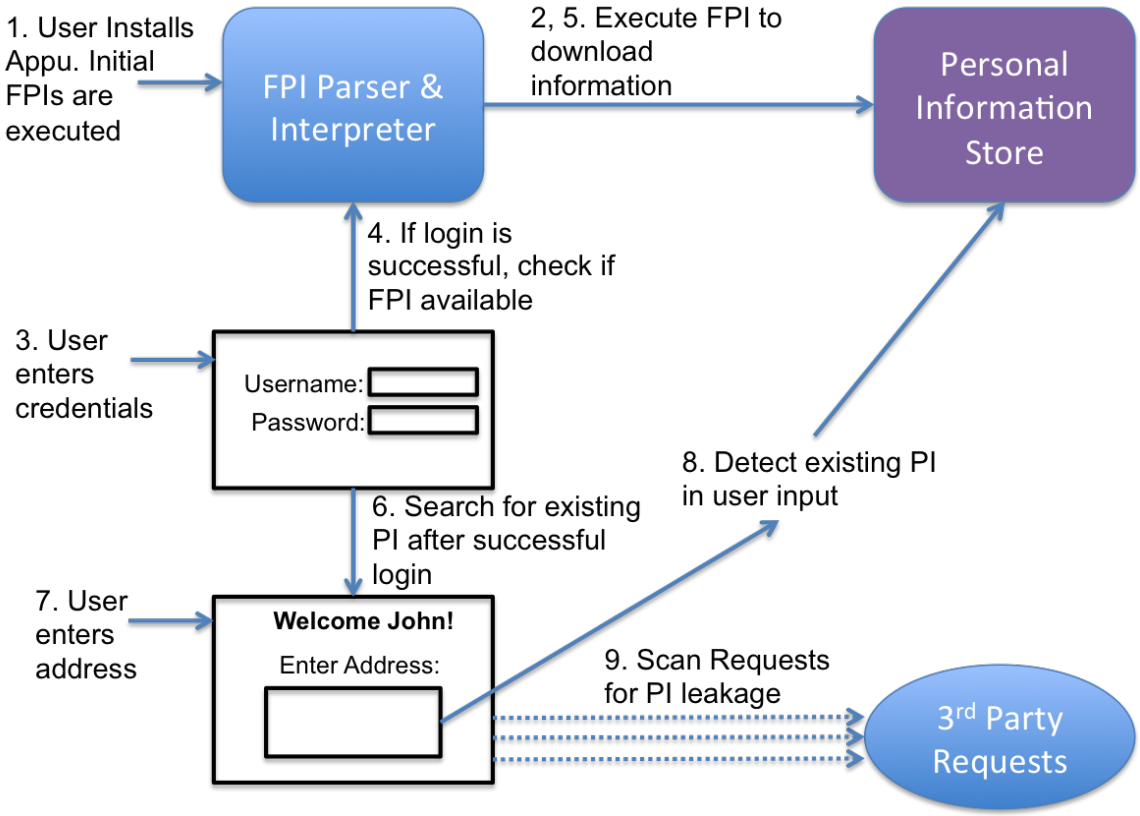


Figure 11: Operational view of Appu: Monitoring personal information spread in the user's browser

detecting that URL is beyond the scope of this work.

3.5 Implementation

We implemented Appu as a Chrome browser extension. Our code base is approximately 15K lines of Javascript, HTML, and CSS. We wrote an additional 5K lines of FPI scripts to actively download account information, taking the total code base to 20K lines.

We faced quite a few engineering challenges while implementing Appu. We will briefly mention an interesting one. When an FPI interpreter executes an FPI script, it opens a slave tab to navigate through the user account. FPI commands are executed on the HTML document present in this slave tab. All the action commands at a specific depth in the FPI tree request for the slave tab at the same time to execute

their action. However, for correct execution, the actions must be processed in depth-first order. Thus, while a node's children are executing their actions, its siblings must wait. To implement this, one needs something like a wait queue data structure or at the very least a mutex structure. Since Javascript does not have either, we invented our own wait queues by manipulating our extensions background HTML page. An FPI child waiting to get access to the slave tab creates an HTML node in the background page. Children on subsequent levels add nested "waiting" HTML elements. Thus, when a particular branch of execution is finished, its next deepest sibling child is selected for execution. This continues until the entire subtree of "waiting" HTML elements is unwound completely, at which point the FPI script is completely executed.

3.6 User Recruitment

We conducted two user studies with Appu. In the first trial, we had less features implemented as compared to the second. In the following section, we describe which features were implemented for each trial. We will describe the results from each trial in the subsequent section.

3.6.0.1 Trial 1

We deployed our first version with a significantly smaller number of features on 200 Amazon mechanical turk users. We also deployed it on additional 40 users recruited from friend and family member circles. This version mainly had features such as password strengths estimation, password reuse monitoring, and FPIs execution. It also collected metadata about fields on a website where user typed information in. This metadata consisted of the length of the information typed and the name of the input field. In addition to that, when a user uploaded a file, Appu also logged that event but not the file type nor its size. It could not detect PI spread passively. It also could not check if a password is present in password dictionaries in the possession of

attackers. It also could not detect different user accounts on the same site.

Data collected by Appu during this deployment was converted to metadata and then anonymously reported. Thus, we could tell that some user with a password having an estimated strength of 12-bits, was reused in Google and Reddit. Similarly, personal store population using FPI would be reported as for user "X," as three different credit card numbers were downloaded from Amazon.

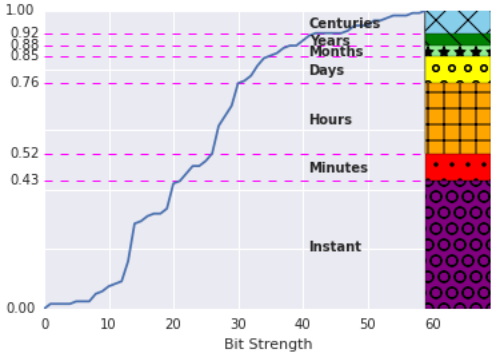
It collected information text characters typed by the user on certain sites, and the name of the field where this data was typed into. By analyzing the input field's name, and the length of the characters typed in, and with external investigation, we were able to identify users submitting various types of PI values such as social security numbers, credit card numbers, addresses, names, phones etc.

3.6.0.2 Trial 2

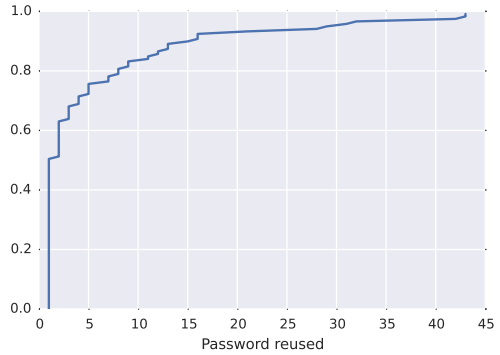
We conducted another trial with all the advanced features mentioned in earlier sections implemented with 25 users from November 2015 to March 2016. These users were mainly recruited from a group of friends and fellow students from our university. In addition to that, a handful of users were also recruited from Mechanical Turk. This study is cleared by 'Institutional Review Board For Human Subjects' at Princeton University (Number: IRB #0000007470). We present interesting findings from these deployment in the next section.

3.7 Results

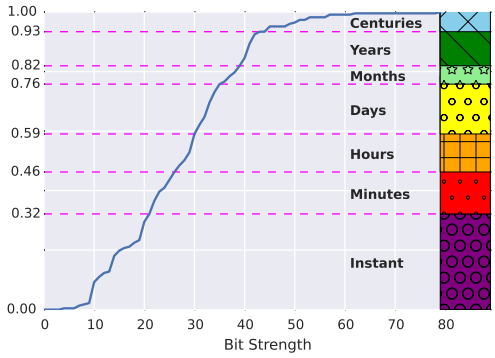
In this section, we discuss results from both of our deployments. We find that users often protect important personal information with insecure passwords. When Appu warned users about changing their passwords to improve their security, users did so only in the cases when they could immediately act on the warning because the password modification links were easily accessbile.



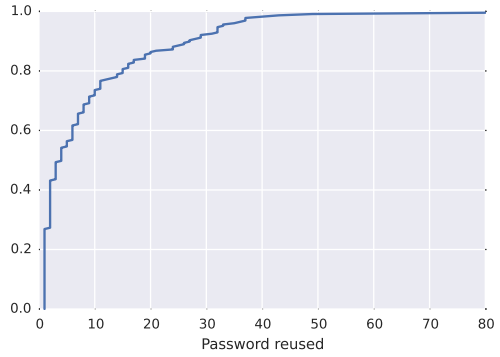
(a) SSN Password strengths of accounts where users have submitted social security numbers. As one can see, 76% of passwords can be cracked merely in few hours using brute force where as 85% can be cracked within few days.



(b) SSN Password reuse across sites. About 60% of passwords are not reused. However, about 20% passwords are reused across as much as 8 sites, whereas rest of the 20% passwords are reused on more than 8 sites, sometimes even up to 40 accounts.



(c) CCN Password strengths of accounts where users have submitted credit card numbers. As one can see, 59% of passwords can be cracked merely in few hours using brute force where as 76% can be cracked within few days.



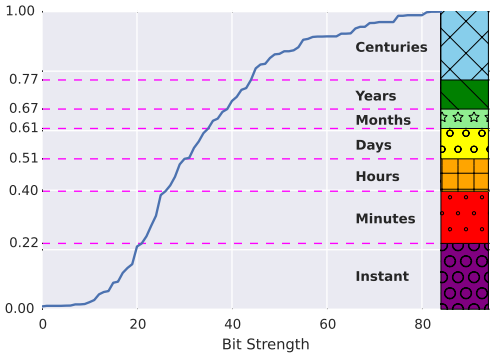
(d) CCN Password reuse across sites. About 25% of passwords are not reused. However, about 35% passwords are reused across as much as 7 sites. Next 20% passwords are reused on as many as 15 sites. The last 20% of passwords are reused in as many as 80 sites.

3.7.1 Trial-1 Results

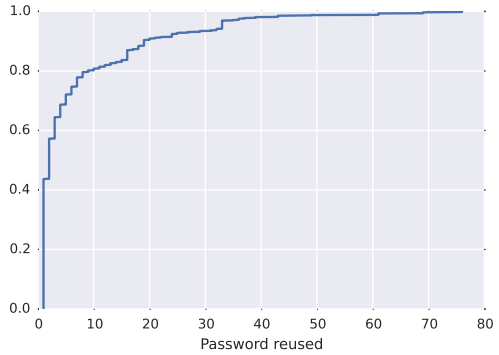
In the first deployment with 240 users, we found that the users logged into 9,227 accounts in 3,963 different websites.

3.7.1.1 Classification of Sites into Security Accounts

We were able to identify 912 accounts where users had submitted at least one of the following important personal fields: social security number, credit card number, address, phone, name, emails etc. Out of these, 143 accounts needed an immediate password change because the password strength was too low. Even though there



(e) File-Uploads Password strengths of accounts where users have uploaded files. As one can see, 51% of passwords can be cracked merely in few hours using brute force where as 61% can be cracked within few days.



(f) File-Uploads Password reuse across sites. In 1346 instances, files were uploaded on sites where password is not reused which is about 42%. For next 38% of file uploads, password was reused in less than 10 sites. For rest of the 20% instances, password reuse was between 10-80 sites.

was a lot of password reuse of the passwords used in these 912 accounts, we cannot determine the security classes of accounts on which they were reused due to lack of passive monitoring in the first trial. However, we were able to confirm that 31 such passwords were accurately reused because users shared equivalent information in each case, where the information would be once again one of the fields mentioned at the start.

3.7.1.2 Social Security Numbers

Fourteen users allowed our FPI to run on PayPal.com, and we could thus download 14 verified social security numbers. However, from the analysis of field names such as "*ssn*" or "*social*," we noted that 87 users have typed in their social security number(fully or last 4 digits) 263 times on 170 different sites. We manually verified that these sites indeed actually ask for social security numbers.

Twenty sites out of these 170 sites are no longer functional. Out of these twenty sites, on 14 sites, users submitted entire social security numbers, where as only the last 4 digits on remaining six sites. Since the sites are no longer functional, it is not clear what happens to the submitted social security numbers. As a result, 12 users could have lost their entire social security numbers to these sites.

Of the 150 still functioning sites, users have given complete SSN on 108 sites and only the last 4 digits on 48 sites. These 150 sites can be mainly categorized into finance(69), government(18), eCommerce/online business(33), and miscellaneous(30).

The top 5 sites where users have submitted their social security numbers are annualcreditreport.com(7), studentloans.gov(7), protectmyid.com(4), chase.com(4), irs.gov(4).

Out of the 173 times when the users have entered their full SSN on some site, we found that 44 number of times, the user did not even login to the site. A maximum of 5 users have entered full SSN without logging in on annualcreditreport.com. This make sense as this site is mostly for checking credit reports.

Figure 12a and 12b show the CDF of password strengths and the amount of password reuse used on sites where a full SSN was typed by the users.

3.7.1.3 Credit Card Numbers

97 users allowed our FPI to run on 15 sites, and Appu was able to successfully download 130 credit card numbers and corresponding information such as the user's real names and addresses.

However, from the analysis of field names such as "*ccn*" or "*credit*", we noted that 102 users have submitted their credit card information 644 times on 237 distinct sites.

Out of the 644 times when users provided credit card information to a site, only 478 times it was submitted to sites where users have an account. That means, 166 times, on 95 different sites, users submitted credit card information without ever opening an account. The top 3 sites where users have been giving out credit card information without opening an account are delta.com, priceline.com and papajohns.com. We verified that one can indeed enter credit card information without opening an account on these sites.

Of the 478 times when users entered the CCN information on sites where they have an account, if we ignore multiple instances of the same user entering credit card information on a particular site, then we get 227 unique user and site associations.

Figure 12c and 12d show the CDF of password strengths and amount of password reuse used on sites where complete CCN information was typed by the users.

3.7.1.4 Uploaded Files

Appu recorded total 4,102 instances of files getting uploaded across 428 different sites by 152 users. In 626 instances, users uploaded files without opening an account on the site.

Figure 12e and 12f show the CDF of password strengths and amount of password reuse used on sites where these 4,102 files were uploaded.

3.7.2 Trial-2 Results

In the second deployment with 25 users, we found that the users logged into 211 accounts in 103 different websites. We observed that in all, users used 78 different passwords.

We first downloaded users' personal information from `Google`, `Amazon`, `LinkedIn`, `Paypal`, and `Facebook`. Then, we detected the spread of this information across sites by passively observing the presence of this information in a)loaded webpage, b)user input, c) set cookies, and finally d)third party requests. Figure-12 shows personal information spread for 25 users across 211 accounts. The figure does not show two additional ïňĄelds that we monitored which were "Credit Card Numbers" and "Social Security Numbers," because their presence was low. We detected presence of "Credit Card Numbers" on 8 webpages. We also detected that users had entered "Credit Card Numbers" twenty three times. We detected presence of "Social Security Numbers" only twice on webpages.

Based on this personal information spread and using the model presented in

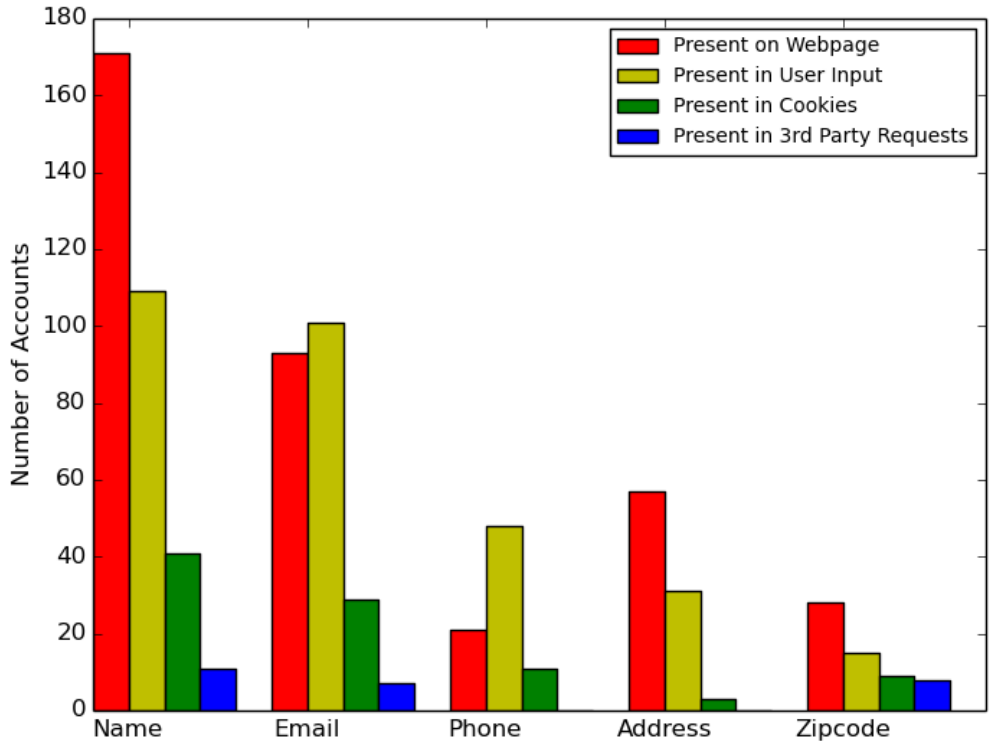


Figure 12: Personal information spread for Appu's users' across the web.

section-3.3, we classified user accounts into three categories. We then checked if the users were using passwords across categories. Fifteen passwords were reused incorrectly across the three categories of accounts. Nineteen passwords belonged to already cracked passwords detected using bloomfilter based mechanism described in section-3.4.2.3. Seven of these 19 cracked passwords protected confidential information and needed immediate change. We detected that 27 passwords had very low password strengths(estimated instant cracking time using password crackers), yet they protected either confidential or private information. Due to these reasons, accounts needed 43 immediate password change. We showed users 29 warnings suggesting password change. However, users only changed 7 passwords. This low rate of password change could be because of engaging a user at an inconvenient time, as well as not providing exact links to change the password. We show a summary of

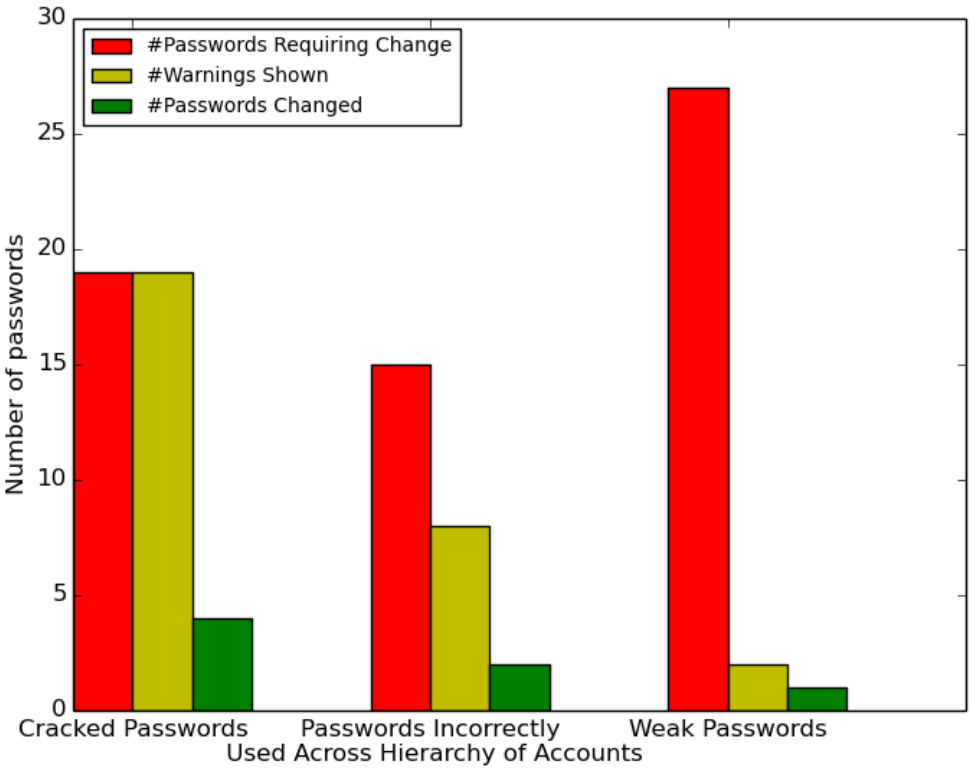


Figure 13: Measuring Appu's effectiveness by counting modified passwords by a user. Left most column shows number of passwords that should be changed. Middle column shows how many password change warnings were shown to the user and the rightmost column shows how many passwords were actually changed by the user.

password related detections, warnings shown, and changed passwords in Figure-13

3.8 Discussion

In this section, we discuss some of the limitations of Appu and possible ways to overcome them. Appu is dependent on `FPI` to seed the sensitive user database as a starting point. Thus, it is required that FPI should be available for sites where a user has an account. In practice, we found that most users have an account on top sites such as Amazon, Google, and Facebook, and hence can be guaranteed to start with seed data. However, sites themselves might change the layout or rearrange the account information. In such cases, the `FPI` will be broken and has to be fixed

manually. Finally, on sites where `FPI` is not available, and an account has already been made, discovering the sensitive information only happens when a user visits the page containing the information, and when that information is already present in the seed database. In the same way, any files or text typed in before Appu was deployed cannot be detected in a generic way.

Detecting when a user is logged in 100% correctly is hard. The only right solution is to check for presence of authentication cookies necessary to access the account. However, in practice, we found that most sites display the username whenever she is logged in and since often usernames are shared across sites, it would be matched against a username in the seed database.

Detecting an account's importance from the perspective of the user is hard because often a user himself might be unaware. For example, news outlet and forum sites such as Reddit is ostensibly unimportant for most of the users. Even though a user may share unique anecdotes on Reddit, she may wish to do so anonymously, and would not want her friends and family members to find out her username, making that account important for the user. As mentioned in section 3.3, we have used a simplistic heuristics to decide the site importance purely based on the quantity of the data. Additional automatic detection may require user input and hints.

We do not consider malicious Javascript as an attack vector at the moment. However, if a third party JS is included on a webpage where sensitive data is present or submitted by user, then such JS can just as easily leak this information to attacker.

3.9 Summary

Appu is limited in certain ways since it works as a browser extension. Most users these days are also heavy smartphone users. In fact, it may happen that users use a specific webservice heavily only via smartphone. Thus, Appu will miss all the data shared on devices without browser extension support and also will not capture

signals indicating the site importance. A future version of Appu that will monitor data exchange occurring via such devices.

Appu does not provide any intelligent search on the basis of keywords about the uploaded data. Adding such support would mean keeping a database of typed in text as well processing and extracting keywords from uploaded files. Due to the recent advancement in Javascript ecosystem such as Node and Emscripten, it would be somewhat easier to reuse existing libraries to process files in various formats.

Finally, we also need to add support to analyze all the third party Javascripts included on a webpage in the eponymous context. One would also require a way to certify each third party Javascript, warning the user in the absence of certificate and blocking communication with third party domains.

CHAPTER IV

BETTER SESSION CONTROL

4.1 Introduction

The web's core protocol, the Hypertext Transfer Protocol (HTTP), is inherently stateless. To manage higher-level application states, web applications commonly store information about user sessions in "cookies." In addition to maintaining the user's state, websites also use cookies to *authenticate* users. A user who initially logs into a website using a username and password could subsequently simply present *authentication cookie(s) (auth-cookies)* for access to that site. Once a user authenticates to a web server, the auth-cookies are thus a critical security linchpin: In many cases, access to these auth-cookies gives an attacker complete control over a user's account [121].

Existing guidelines define the syntax of protecting cookies using various flags and security protocols. However, they fall short of detecting the exact set of cookies to which this syntax should be applied to.

Modern web applications are complex. They have millions of lines of code, and often use legacy code from other components or libraries. Today's web sites and applications are multi-faceted, so a "login" or "authentication" is no longer strictly binary. For example, a user may have different levels of authorization or access to different parts of a web site (*e.g.*, a user can view account balances but not execute trades, or add items to a shopping cart but not purchase items or ship to a different address). For such complex applications, ensuring that authentication logic is correct can be a challenging task—one that, as we have discovered, web developers often get wrong. For example, in May 2014, the incorrect security settings of WordPress's

auth-cookies left users vulnerable to session hijacking attacks [34].

Our results show that these problems are not isolated: We found 113 different sites—including 62 Alexa top-200 sites—that were vulnerable to attacks where an attacker could steal auth-cookies to gain unauthorized access to parts of a site or its entirety. (In 10 cases, were able to ïňĄnd a way to report the vulnerability to the site operators and did so.) We also evaluated popular eCommerce frameworks—Magento [90], BigCommerce [38], WooCommerce [135], and Volusion [132]—which are used in building millions of eCommerce sites; more than 900,000 sites that use these off-the-shelf frameworks may still be vulnerable to session hijacking attacks. In some of those cases, such as Vimeo and BigCommerce, the developers have confirmed and fixed the bugs that we have found; Amazon is in the process of fixing the bug that we reported to them.

Finding these vulnerabilities highlights an important and prevalent weakness in today's web authentication frameworks. Finding the vulnerabilities was a challenging problem in its own right, since we generally cannot assume access to the source code of many web applications. Indeed, even a developer who has complete access to source code may find the analysis challenging, due to the complex nature of these applications. Thus, we needed to design a black-box approach to detect authorization cookies for a web application. We developed a tool called *Newton* that developers can use to help them more easily identify these vulnerabilities. We present the following contributions:

- We suggest improvements to existing best practices for securely implementing auth-cookies. We assembled a 6 rule checklist for secure auth-cookies implementation by making 5 of the existing rules precise, while adding one more of our own.

- We developed a tool, Newton, an easy-to-use Chrome extension. Anyone can

use Newton to identify the auth-cookies required to access a portion of a website and analyze if these auth-cookies are sufficiently protected using the security checklist.

- Using Newton, we conducted a security audit of 149 sites, with the help of our pilot users, and found previously unknown vulnerabilities in 113 sites, including Yahoo, Vimeo, and BigCommerce. We have reported these problems, and many of them have either been ïňĄxed already, or been acknowledged as an existing problem (which sometimes has been attributed to a third-party library).

4.2 Modern Web Authentication

We present general background on web cookie operation, introduce a model for how *auth-cookies* authenticate users on websites today, and describe scenarios where these cookies might be stolen or otherwise compromised.

4.2.1 Formal Authentication Model

When a user visits a website, the web server sets *cookie* properties (*i.e.*, name, value, flags, expiration date, and a match rule) either statically using `Set-Cookie` or with JavaScript. When a user makes an HTTP or HTTPS request, the browser sends cookies whose match rules correspond to the requested URL and protocol. A browser will not transmit cookies with the HTTPS attribute over a regular HTTP connection.

A web server sets multiple *auth-cookies* when a user initially authenticates to the web server. Auth-cookies ensure that a user's session remains continuous in case connectivity is interrupted (*e.g.*, due to termination of a TCP session or a change in the user's IP address). Many websites use different combinations of auth-cookies to control access to different parts of the site (*e.g.*, a user filling a shopping cart might require a different set of cookies than purchasing the items in the cart).

We can formally represent a user's logged-in state to any part of a site as a disjunctive normal form (DNF) formula over cookies. Any one of the multiple conjunctions of auth-cookies (each with the correct value that matches exactly) can allow a user to access that part of a site. For example, to access dating site "match.com," the browser must satisfy DNF: (Handle AND Password) OR (SECU). Note that is not a generic version of DNF, because every site requires at least one auth-cookie to authenticate a user.

We call each "one or more conjunctions of one or more literals" a DNF term. Calzavara *et al.* refer to them as authentication tokens [48]. Whereas the related work only focuses on authentication cookies used to login to the main part of the site, we discovered that often sites have multiple parts controlled by different DNFs. A user's logged-in state on each part of the site is decided by relevant DNF for that part. Furthermore, a site may have login dependencies, such that logging into (or out of) one part of a site results in automatic login (or logout) to other parts of the site. We can represent those dependencies as a directed graph, where a directed edge from A to B means that the login state of B depends on the login state of A.

We aim to discover (1) the DNF formula for authenticating a user to each independent part of a website; (2) based on the DNF formula, the dependencies between each part of the site, if any exist. Once we have recovered this structure, we can determine whether any combination of cookies could be stolen by an attacker, thus resulting in unauthorized access. (Section 4.2.2 discusses our threat model for an attacker.)

Example Suppose that when a user logs into `yahoo.com`, the site sets the `Y`, `T`, and `SSL` cookies. The `Y`, `T`, and `SSL` cookies match on the domain `.yahoo.com` and path("/"), but the `SSL` cookie only matches on HTTPS connections to `.yahoo.com`. Different combinations of cookies (and their correct values) authenticate the user to

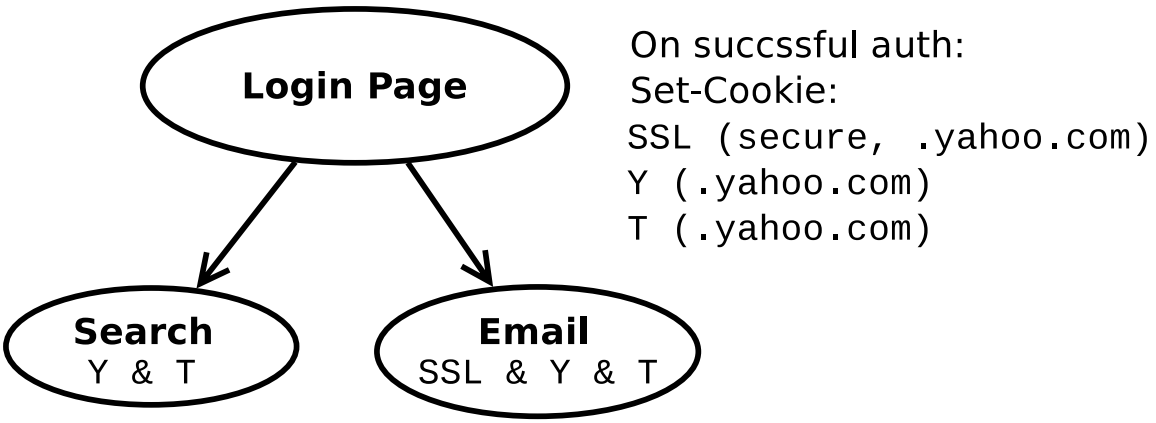


Figure 14: A simplified authentication model for `yahoo.com`.

different parts of the site: `Y` and `T` allow a user to access search history; accessing email requires the `SSL` cookie, as well.

Figure 14 formally represents the web authentication model for this simplified example of the login process on `yahoo.com`. A successful login automatically results in login to the search and email portions of the website (and the corresponding cookies being set). Each part of the website has a corresponding DNF login formula, as shown within the node. If a user logs out of the "login" portion of the site, the user is logged out of both the search and email portions of the site, as represented by the dependency graph. However, a user can be logged out of email part yet remain logged in on the search portion of the site.

4.2.2 Threat Model

As discussed earlier, once a user is logged in, auth-cookies become a security linchpin. If an attacker can steal these cookies, then she can hijack the user's session. In this section, we list a couple of different ways in which a victim's auth-cookies can be harvested.

First, we are concerned with the threat of an on-path attacker who intercepts unencrypted auth-cookies. In this case, a vulnerable DNS server would allow an attacker to poison its cache and reroute a victim's traffic to machines under the

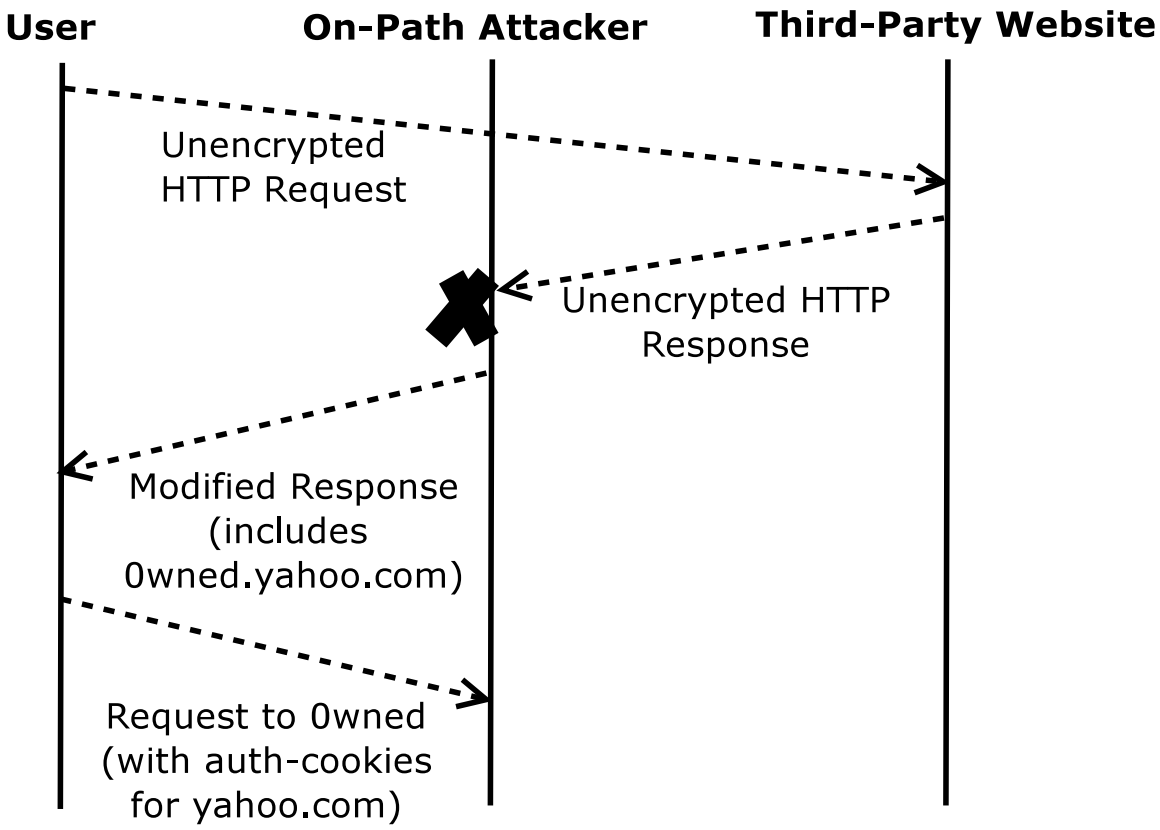


Figure 15: An on-path attacker can gain unauthorized access to a user's search history. (We discovered this vulnerability on `yahoo.com`.)

attacker's control [35]. An on-path attacker or any attacker who can control and manipulate DNS traffic destined for a user's machine—as is the case with untrusted WiFi hotspots, DNS servers that are vulnerable to cache poisoning, or rogue ISPs—can induce a user's browser to submit all auth-cookies without the `secure` flag across *any* site where the user is logged in.

Figure 15 illustrates this attack. In this case, the attacker is the administrator of the hotspot or has otherwise compromised it, perhaps using off-the-shelf tools [66, 97]. Suppose that the victim is logged into a Yahoo account and always accesses `search.yahoo.com` over HTTPS. The two cookies that allow access to a user's search history, `Y` and `T`, do not have the `secure` flag set. Thus, if any other HTTP webpage that an attacker controls injects content from a `yahoo.com` subdomain, the browser will leak the auth-cookies to the attacker's machine.

Second, if a website has other vulnerabilities, such as cross-site scripting (a common vulnerability [104]), an attacker may also be able to steal a user's auth-cookies that are not protected with the `HttpOnly` flag. Third, a user may log into a site from an untrusted device. Then even if the user logs out of the site, if the site does not invalidate auth-cookies, then the administrator of that untrusted device could take over the user's session. Fourth, a user's personal device may be stolen. If the user is already logged into a couple of different services in a browser on that device, and the site does not implement auth-cookies security correctly, then the sessions may remain valid, even if the user changes his or her password.

4.2.3 Limitations of Existing Defenses

We now will explore existing defenses against session hijacking via cookie stealing, as well as the limitations of these defenses.

Cookie flags To protect cookies against theft by untrusted third parties, each cookie also has two boolean attributes, `HttpOnly` and `Secure`. Javascript running in the user's browser can read, write, and delete cookies created by the same domain [37]. If the cookie's `HttpOnly` flag is set to true, then even JavaScript from the same domain cannot read that cookie, which thwarts potential cross-site scripting attacks [70]. If a cookie's `Secure` attribute is set, then the browser will only return the cookie over HTTPS.

One might think that perhaps all cookies involving authorization should have the `secure` and `HttpOnly` attribute set, but code complexity can make it difficult for a developer to identify the set of auth-cookies that are used for some portion of a website. Additionally, setting `HttpOnly` on all cookies is not practical because some JavaScript functions legitimately require access to these cookies. One example is "double submit cookies" [106], a mechanism to prevent cross-site request forgery whereby a web browser sends a cookie with some value in the HTTP header, and the

JavaScript code sends the same value in the HTTP request body. This mechanism requires the JavaScript to access a cookie value, so setting `HttpOnly` is not possible. Setting `secure` on all auth-cookies is not practical, either, because an application may serve content over both HTTP and HTTPS.

HTTP Strict Transport Security (HSTS). HTTP Strict Transport Security (HSTS) defends against certain man-in-the middle attacks by forcing some web requests to upgrade from HTTP to HTTPS. If the server sets the HSTS policy such that an auth-cookie-domain is a subdomain of the HSTS policy, then the browser would always send the auth-cookie over HTTPS, even if the `secure` flag is not set on the cookie. HSTS could overcome situations where a developer fails to set the `secure` flag on an auth-cookie, but unfortunately this approach does not always work.

Deploying HSTS requires careful analysis of a site, and even reorganizing website content into different subdomains. In some cases, reorganizing a site to use HSTS without breaking the site's functionality is not even possible [74]. HSTS is still not widely deployed. According to a report in November 2014, only 3,406 sites out of approximately 150,000 popular sites had deployed HSTS [124]. Even if HSTS is deployed, it may be vulnerable to SSL downgrading attacks [69] and is often misconfigured [82].

4.3 Newton: A General Cookie Auditing Tool

To check whether the best practices from the previous section are followed by a website, one has to first identify auth-cookies and their exact DNF required to access a particular part of the site. Unlike Calzavara *et al.* who run their algorithm in a controlled lab setting, we aimed towards four different kinds of users: a) application developers performing black-box testing, b) penetration testers without source-code access identify vulnerabilities in existing web applications, c) security-conscious users identify when they might be vulnerable to session hijacking attacks, and d) as a crowd

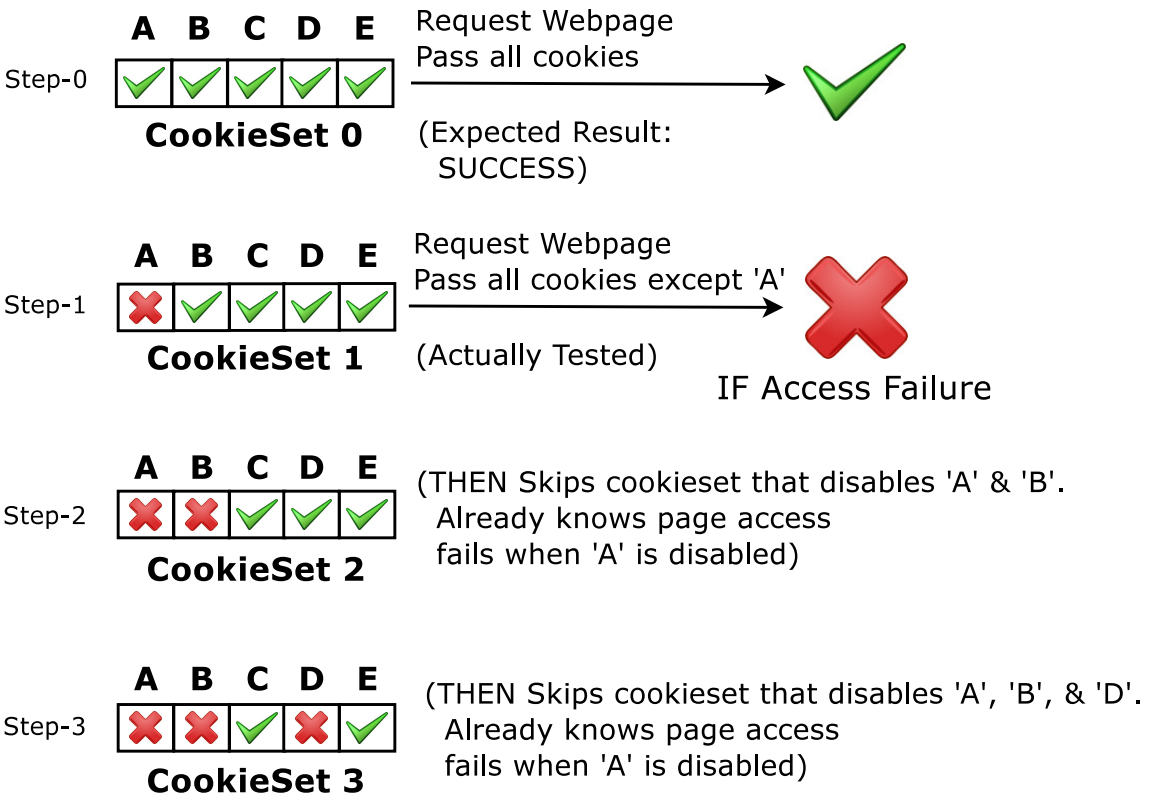


Figure 16: Suppose that $(A\&B)$ is the auth-cookies combination that Newton is trying to discover. A page fetch request will not succeed if cookie A is not present in the cookie-set. Newton will conclude that A is *part* of the auth-cookies combination; it does not need to test other cookie-sets that do not contain A.

sourcing platform where users can run the tool and find DNFs on different sites with minimal configuration.

In this section, we first list the similarities and differences of our algorithm from previous work. Then we list the specific goals that are different due to our decision of running the tool in non-controlled setting.

4.3.1 Basic Algorithm

Newton identifies a website's auth-cookies using the following approach:

1. Retrieve a webpage without suppressing any cookies,

2. Disable a combination of cookies and then retrieve a webpage with rest of the cookies enabled.

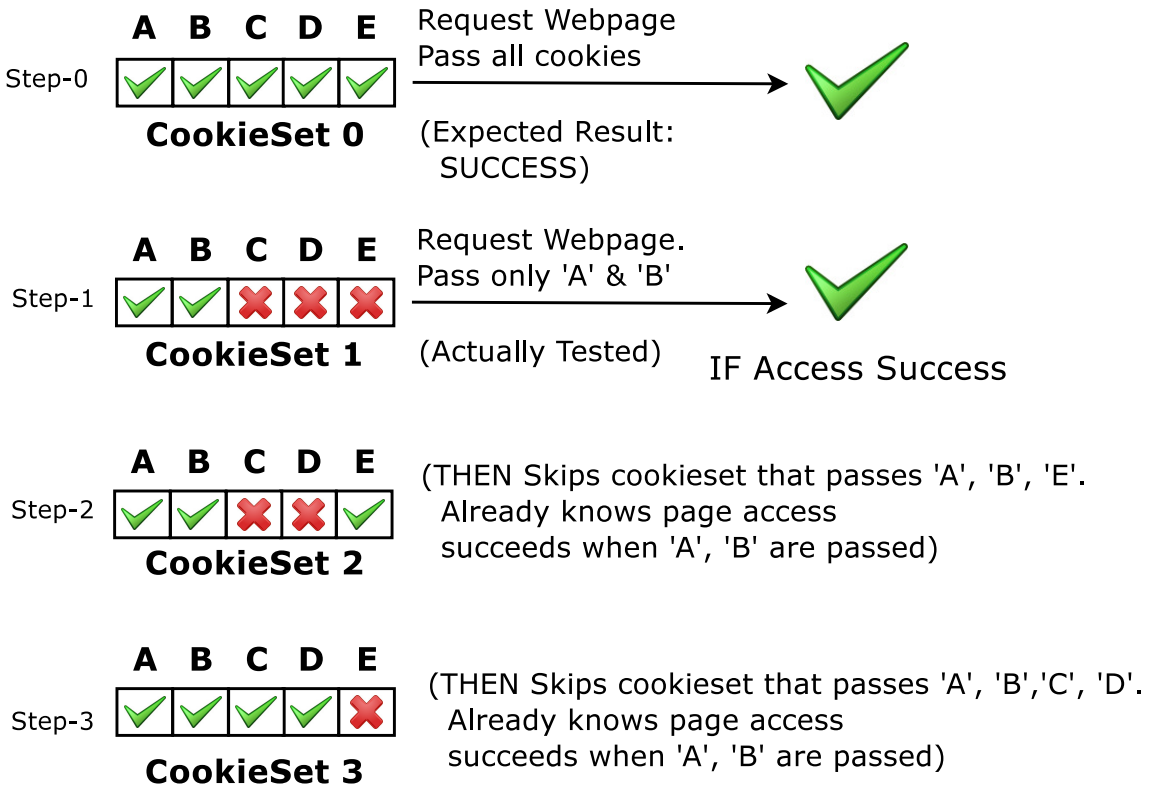


Figure 17: Suppose that $(A\&B)$ is the auth-cookies combination that Newton is trying to discover. A page fetch will succeed because both A and B are included. Newton concludes that either A, or B, or both are *part* of auth-cookies combinations; it does not need to test other cookie-sets with A and B enabled.

3. Compare the response page with the one fetched in Step 1.

4. If the webpage is not retrieved successfully, then we mark the combination of disabled cookies as required for accessing the webpage.

This approach faces two major challenges: (1) there are exponentially many possible combinations of cookies to test; (2) the unique design of each website makes it difficult to develop tests and draw conclusions about success or failure—even determining whether a user is logged in can be surprisingly challenging, for example.

4.3.2 Challenges

In this section, we elaborate on the two major challenges presented above as well as how Newton addresses them.

4.3.2.1 Efficiently Testing Cookies

At first glance, it might appear that if a website sets N "login cookies," then Newton would need to send HTTP requests to the website with all 2^N possible cookie combinations to determine all sets of cookies that represent auth-cookies. Fortunately, we can use the outcome of some cookie-set tests to infer the outcomes for other cookie-sets, precluding the need to test all sets with HTTP requests. Figure 16 shows one such inference optimization: if Newton determines that the user is logged in when all cookies in the cookie-set are sent, but that the user is logged out when cookie A is not set, then Newton can conclude that A is *part* of an auth-cookies combination. Conversely, if Newton determines that a user is logged in when all cookies in a cookie-set are sent, and also that the user is logged in when *only* cookies A and B are sent, then Newton can infer that any cookie-set containing A and B will succeed, and either A or B or both of them are auth-cookies. Figure 17 shows this optimization. These optimizations are similar to pruning techniques used by Calzavara *et al.*. Their DNF terms generation mechanism is faster than ours due to using the Apriori algorithm [32]. In spite of using suboptimal DNF term generation, our auth-cookie computations typically finish within five minutes. Incorporating the Apriori algorithm into Newton is a straightforward extension.

4.3.2.2 Running Newton in Non-Controlled Settings

Instead of guessing auth-cookies for the main part of the site—as is the state of the art in existing approaches—it is imperative to compute the *exact* DNF required for critical parts of a site. Interestingly, this task actually is easier if we collect data from when a user is naturally browsing the web, rather than in a controlled setting. The main idea is that Newton automatically detects the DNFs across multiple parts of multiple sites and compiles this information at a central place for various uses. This design makes it much easier to gather more data at scale, but requires Newton to run

in the background on real users' machines, which requires solving the challenges that we describe in the rest of the section.

4.3.2.3 Deciding Which Portions of a Site to Test

To exhaustively enumerate all DNFs for a website, the tool needs to know all critical parts of the site. This task is easier when a site developer or penetration tester explicitly identifies different logical parts. Yet, if this task is performed by a crowd sourcing platform, the tool must automatically identify critical parts of the site and execute tests on each portion of the site independently. In such cases, the tool identifies whether the user has entered a password and logged in to the site recently. The tool may also need to apply some heuristics, such as detecting that some portion of the site is served over HTTPS, or the presence of critical information such as credit card numbers or social security numbers.

4.3.2.4 Detecting Login Success or Failure

Identifying login cookies also depends on an accurate mechanism for determining whether the user is logged in, yet differences in website design can make even this seemingly simple task surprisingly difficult. Like Calzavara *et al.*, we found that a reasonable heuristic for doing so is to determine the presence of a username on the site itself. However, since we wanted our tool to run as a crowd sourcing platform with minimal configuration, we did not want ordinary users to enter their usernames into the tool. For this reason, we wrote scrapers in the tool to automatically download and populate username corpus from popular 70 sites such as Google, Facebook, Yahoo, etc. In our experience, in most cases the usernames from these sites are sufficient to identify usernames on unknown sites. Section-4.5 provides an example of one such scraper for dropbox.

4.3.2.5 Disambiguating Authentication Failures

Finding the correct DNF of auth-cookies depends on correctly identifying whether the user is logged in on the currently loaded page. Unfortunately, various transient conditions can also introduce spurious failures that exhibit similar characteristics to login failures, and we must be able to disambiguate these from actual login failures. For example, a user might log out of a site, or the user's session might time out. To disambiguate these results, Newton must also reload pages with all cookies enabled after getting a positive result to verify that the user is still logged in. If Newton detects that the user has purposefully logged out of a site, it will store the current DNF testing state for that part of the site and resume the computation the next time user accesses that part. Sometimes an authentication failure may occur because of poor network conditions. Newton waits a sufficiently long time for the page to load and attempts reload under poor conditions, to minimize the likelihood of mistaking poor connectivity for a login failure. Newton determines the waiting-time threshold by maintaining a moving average of past successful page load times.

4.3.2.6 Avoiding Disruptions to User Session

Sending a website incorrect cookies can cause the website to log a user out, disrupting the user session. For example, `Facebook` has an auth-cookies combination '`xs` AND `c\_user`', and dropping `xs` resets the `c\_user` cookie and logs the user out, thus disrupting the user experience. We needed to devise a mechanism to send arbitrary cookie sets from a real user's browser *without disrupting the user if we send an incorrect or invalid set of cookies*. To do so, Newton runs all tests using an in-memory set of shadow cookies that mimic the user's real set of cookies.

Because Newton repeatedly loads the same page during auth-cookies computation, one concern is that it could affect user's state on the server side. A page load is an HTTP GET request which is an idempotent operation [61]. It should only have an

effect the first time the user performs the request. It is possible that repeated requests might have unintended side-effects, however (*e.g.*, a user might be rate-limited if the client is perceived to be performing excessive requests). For these reasons, we are cautiously limiting our real world study with the alpha users. Although so far, we have not seen any side effects either in our lab testing or from our users.

Some of Newton's tests of a website's practices depend on interactions with the user that should be as infrequent as possible or are not guaranteed to occur in the first place. For example, to test that a site invalidates a user's auth-cookies after the user has logged out, Newton must discover that the user has logged out, restore the previous session cookies, and test if the user is still logged in. Newton automatically discovers when a user logs out using various heuristics, but to restore the auth-cookies, we explicitly ask for user permission, which becomes annoying if performed repeatedly. Testing whether a website changes the user's auth-cookies when the user changes his or her password requires waiting for the user to first change his or her password. Currently, Newton automatically discovers when a user changes his or her password for a particular account, at which point it subsequently performs testing. Because Newton must wait until a user changes his or her password to perform this test, we cannot guarantee that this test will always be done.

Because Newton retrieves webpages repeatedly while suppressing different combinations of cookies, a concern is that such repeated requests might have unintended or unforeseen side effects. For example, some sites aggressively monitor auth-cookies in the HTTP requests, and if the HTTP request sends fewer auth-cookies than required for successful authentication, the server may log the user out. While this action reflects good security practice, it also makes it more difficult to compute auth-cookies. In the future, maintaining instances of such interference or making auth-cookie computation probabilistic could help mitigate some of these effects.

4.4 Newton: Computing Auth-Cookies

In this section, we describe the design of Newton, a tool that we have built to help website administrators in auditing, and to help common users to protect their web sessions. We refine the basic idea that we presented in Section 4.3.1 to address the various practical concerns that we raised in Section 4.3.2.

4.4.1 Detecting Login Status: Username Presence

Newton uses the presence of the user's name on the webpage to determine that the user is logged in. To detect the presence of a user's name without explicitly asking the user (which is error-prone in any case, since the webpage may display a real name to a logged-in user, rather than the username), we developed a *domain specific language* that allows us to specify how to scrape a user's username and other information from different sites. Using this language, we developed scraping scripts for many popular sites, including `Facebook`, `Google+`, `Amazon`, and many dating sites to determine a user's real name and username.

Determining a user's login status on a broad range of sites using the presence of a username or real name on the page requires multiple bootstrapping steps. First, we create dummy accounts on an initial "seed" set of sites to help us codify the relationship between the site's structure and the location on the site where the user's name resides. Knowledge of these relationships allows us to develop scripts in our domain specific language to scrape real names and usernames from these sites for any user who installs Newton. We then ran Newton for each webpage for each of the sites where we could determine these relationships. Newton then determines the auth-cookies for each of these webpages (using the algorithms that we describe below). After the user installs Newton, the tool can determine whether a user is logged in on these sites by examining whether the appropriate auth-cookies are set; it can then determine the user's name (or username, depending on the site) by scraping the page.

Finally, with knowledge of the user's name(s) and username(s) from these seed sites, Newton can then determine whether a user is likely logged in on other sites (*e.g.*, banking sites) simply by checking for the user's name (or username).

Using our domain specific language, we have created 70 scraping scripts for top sites where users have accounts usually such as social networking, mailing, utility sites. Using these, we are able to comprehensively cover different variations in a specific user's usernames and thus detect these usernames on sites, even on sites for which we have not written a scraper. However, even if we fail to detect some user's username on some site, Newton can *still* determine the auth-cookie combination for that site by performing the auth-cookie computation for a user who we are able to identify for that site. Because the auth-cookies that a site uses are the same for every user, Newton can build a knowledge base of auth-cookies for a large set of sites that is then shared across Newton users.

4.4.2 Tackling Combinatorial Explosion

The strawman algorithm in Section 4.3.1 computes auth-cookies by disabling combinations of cookies and determining whether a user remains logged in. Although this approach is simple and effective, it does not scale in practice, because the number of tests is exponential in the number of cookies that a site sets, and many sites set tens or hundreds of cookies; once the number of cookies on a site exceeds about 30, testing every combination becomes prohibitive. For example, `live.com` sets 27 cookies at login time. Assuming a single test requires about 200 ms, testing all combinations of only those cookies would require nearly a year. In this section, we discuss how to avoid the exponential cost of testing every possible cookie combination. Although the search space remains exponential, Newton can use knowledge from previous results to avoid testing certain combinations of cookies.

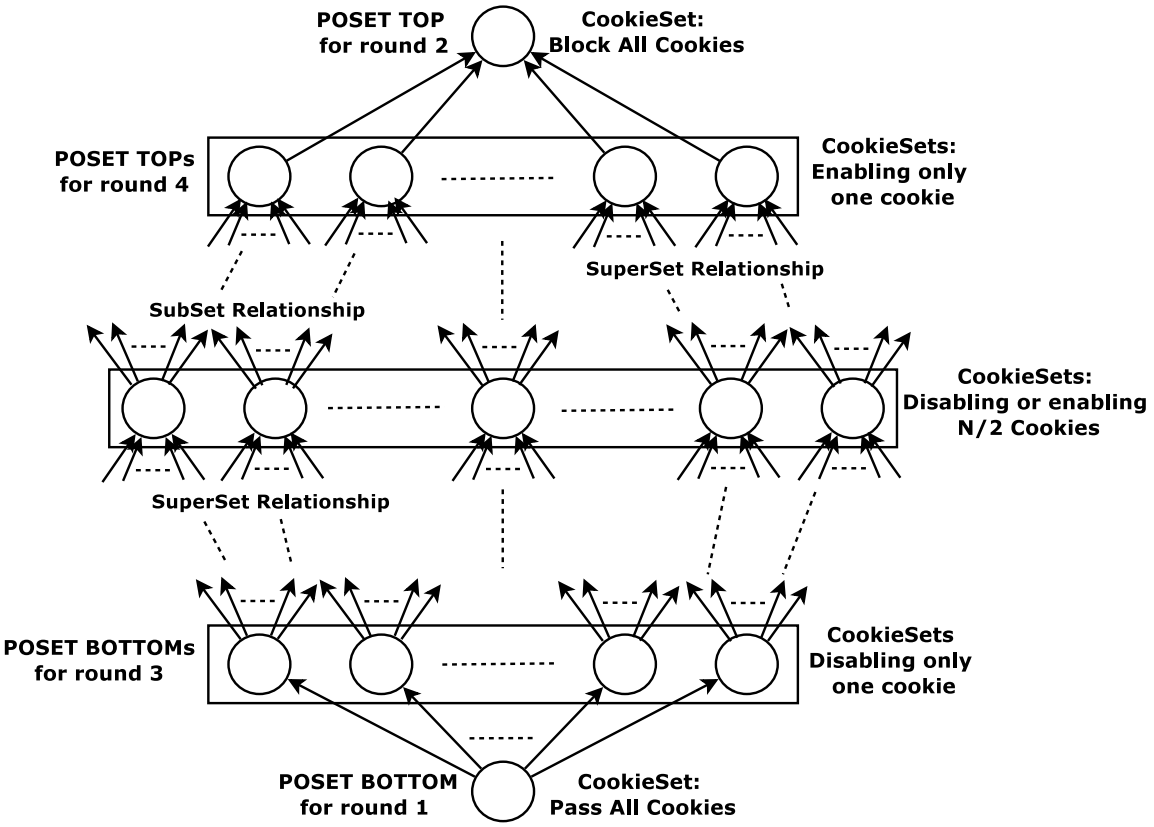


Figure 18: Bounded complete partially ordered set (POSET) representing the set of all cookie-set combinations that Newton would have to test without optimizations. In each testing round, Newton alternately tests current "tops" or "bottoms" of the POSET.

4.4.2.1 Basic optimization

First, Newton partitions the cookies for a site into two sets: "login cookies" (set during a user's login process) and "non-login cookies" (set either before or after the login event). To reduce testing time where possible, Newton initially assumes that a site's auth-cookies are a subset of the cookies that are set at login (the "login cookies"). As we discussed in Section 4.3.2, some auth-cookies may not be set at login, and Newton may need to expand the set of cookie-sets that it is testing to include additional cookies.

Figure 18 illustrates bounded partially ordered set representing all possible cookie-sets that Newton would have to test, either working downward from the top of the

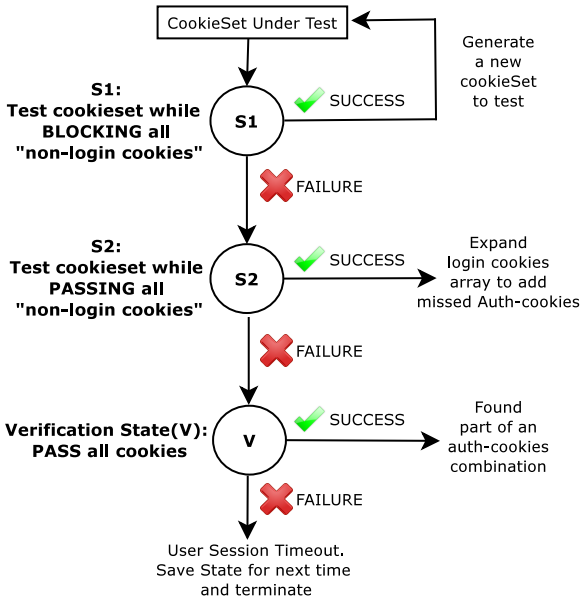


Figure 19: When testing all current bottom cookie-sets(going UP in the POSET), if Newton discovers a negative test result (testing a cookie-set still causes the user to be logged in), then Newton moves on to the next cookie-set to be tested. However, if the result is positive, then Newton performs two additional tests: one to ensure that there are no additional auth-cookies that are in non-login cookie set(state $S2$), and one to ensure that we have not mistakenly assumed a disabled cookies as an auth-cookie combination because a user was logged out during testing(state V).

graph by adding cookies to an empty cookie-set, or by working upward from the bottom by removing cookies from a complete cookie-set. The Newton algorithm alternates by performing one test from the top of the partially ordered set, followed by one from the bottom, removing cookie-sets from the graph as they are determined to either represent an auth-cookie set or not. The algorithm terminates when no cookie-sets remain.

4.4.2.2 Handling corner cases

User logout or session termination during test. When Newton is computing the auth-cookies for a particular site, the user's session may terminate for a variety of independent reasons; a user's session might time out, or a user might initiate a logout. If these events occur during one of Newton's tests, logout should not be attributed to the set of cookies that were being tested at the time the logout occurred. For this

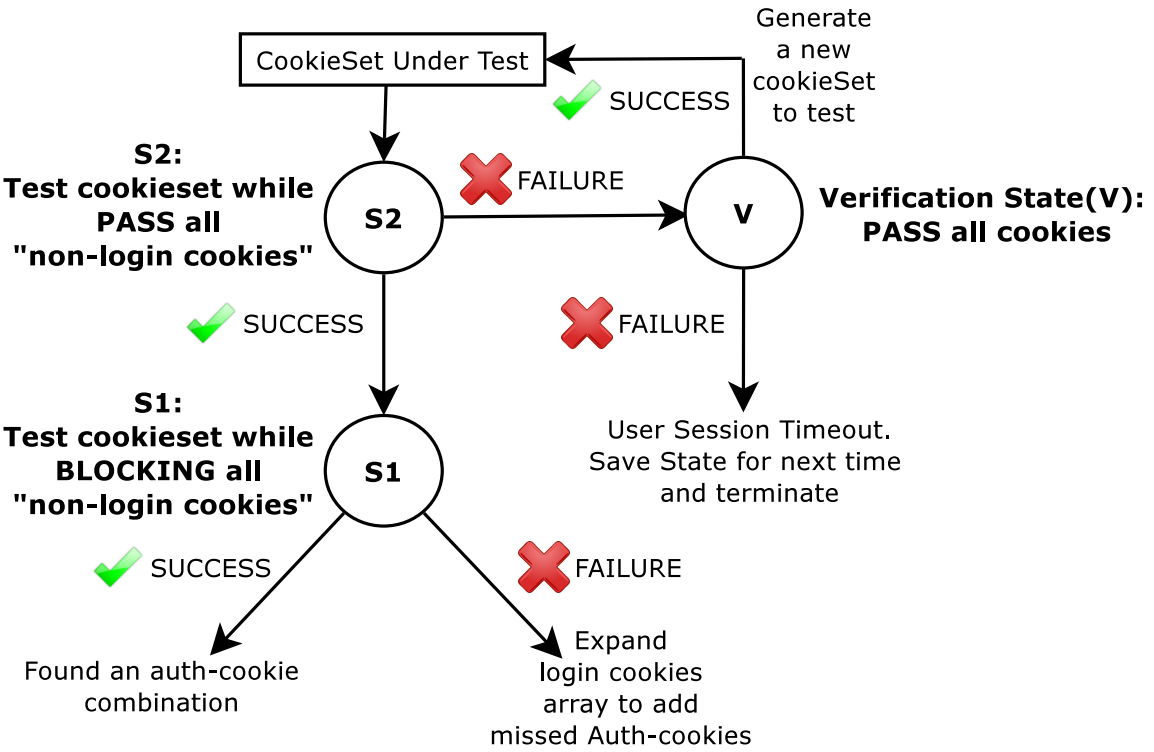


Figure 20: In contrast to Figure-19, while testing all current tops(coming DOWN in the POSET), a negative test result occurs when user is not logged in(expected result with most of the cookies blocked) where as a positive test result is when a user is still logged in.

reason, after the computation of the auth-cookies completes, Newton tests that the user is still logged in by sending all cookies for the site in an HTTP request. State V in Figures 19 and 20 illustrates this logic.

Finding auth-cookies that are not set at login. As we mentioned in Section 4.4.2.1, sometimes an auth-cookie might be set at some time other than when the user logs in, either because the site sets it at some later point after the user logs in (*e.g.*, as is the case with Google Calendar) or because the auth-cookie persisted since the user's previous session. There are two ways to detect that auth-cookies may be set at times other than when the user logs in; states $S1$ and $S2$ in Figures 19 and 20 show the logic to detect these corner cases. For example, in Figure 19, if a set of cookies from the "login cookies" set does not log a user in, but enabling all

of the cookies that are not set at login results in a successful login, then Newton detects that one or more cookies that were not set at login must be part of the set of auth-cookies. In this case, Newton must actually proceed to compute the exact auth-cookies present in the "non-login cookies" set. To do so, Newton constructs a partially ordered set of "non-login cookies" similar to the one shown in Figure 18 to find the additional auth-cookies.

4.5 Implementation

We describe the implementation of Newton and an evaluation of its performance in terms of the page fetches, time, and overall download data required to compute auth-cookies for various sites.

4.5.1 Prototype: Chrome Extension

We implemented Newton as a Chrome browser extension. Our code is open source. Except for the scrapers of site-specific content, all of the code was written in JavaScript; we wrote the scrapers in a domain specific language.

We released a version of Newton to 12 users which were recruited by word of mouth advertisement in our university. We only allowed Newton to compute auth-cookie combinations on a pre-approved list of domains, with the users' consent. Many of the sites that we tested during deployment were not previously tested in the lab. None of these users have reported any performance problems or disruptions as a result of running Newton. As our experience with this initial deployment has been uniformly positive, we plan to release Newton to a wider audience, who we imagine may include both application developers, security-conscious users, and penetration testers.

Figure 21 shows an example of a scraper for `dropbox.com`. The code instructs the browser to go to the settings page for the user (line 3) and locate the HTML tags that correspond to the first and last name of the user (lines 5–14); it then stores the information to local storage (lines 16–21). We have shown one of the simpler

```
<div name=''dropbox''>
<action type=''fetch—url''>
https://www.dropbox.com/account#settings</action>
  <div name=''first—name'' can_be_a_null=''no''>
    <action type=''store'' field_type=''editable''>
      td:contains(‘‘First name’’)+>input
    </action>
        </div>

  <div name=''last—name'' can_be_a_null=''no''>
    <action type=''store'' field_type=''editable''>
      td:contains(‘‘Last name’’)+>input
    </action>
  </div>

  <div name=''email'' can_be_a_null=''no''>
    <action type=''store''
     jquery_filter=''remove—children''>
      td:contains(‘‘Email’’)+
    </action>
  </div>
</div>
```

Figure 21: Example of domain specific language for finding and a user's full name on Dropbox. For sites where we were able to create dummy accounts, we could write modules to infer the user's username or full name from the site, which would then allow Newton to determine if a user was logged in.

examples of a scraper for simplicity; more examples are available on Github [105].

4.5.2 Performance Evaluation

To evaluate the feasibility of running Newton in real-time (*e.g.*, to help users identify threats as they are browsing) as a crowd sourcing platform, we analyzed the time to compute auth-cookies, as well as the bandwidth overhead and number of page fetches required to compute the auth-cookies. We measured the performance of Newton using data from both our field deployment with alpha users and from our own tests, across 149 different websites. Of these 149 sites, 98 were English sites from the Alexa top-200 ranking, where we were able to create a user account. We tested the rest either because our pilot users used them, or because the site themselves were implemented

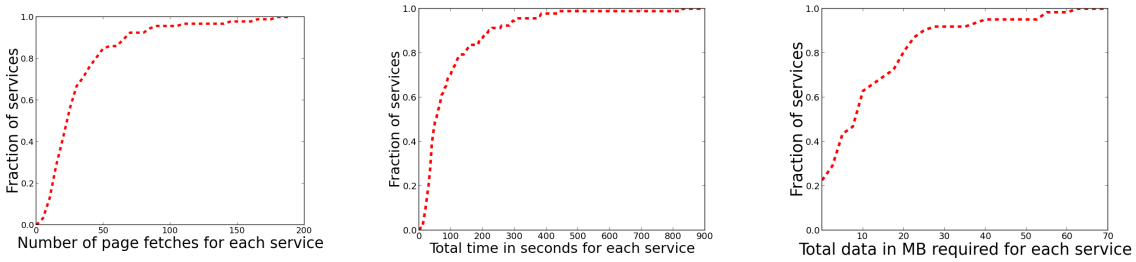
using one of the popular eCommerce frameworks. We conducted these performance tests on a residential network in the United States with a downstream throughput of about 6–7 Mbps. The complete list of sites and gathered results of our analysis are available in a Google document [99]. We focused on testing sites that were either in the Alexa Top 200; popular among our alpha users; have both HTTP and HTTPS content and stored sensitive personal data; or were built using one of the frameworks that we tested.

Figure 22a shows a distribution of the number of page fetches that are required to compute the auth-cookies for a particular site. Newton can compute auth-cookies for 90% of sites with fewer than 100 page fetches; the median number of page fetches to compute auth-cookies for a site is 25. Figure 22b shows that about 80% of these computations require less than two minutes to complete, and Figure 22c shows that about 80% of these computations require the client to download less than 20 MB of data from the webpage for which Newton is performing the auth-cookie computation. These results show that Newton is efficient enough to operate in practice.

Another possible concern is that Newton might introduce excessive server load. As shown in Figure 22a, Newton identifies auth-cookies for most sites with fewer than 100 HTTP requests. We also note that the results of Newton's auth-cookie computations are a one time effort for that part of the site. Additionally, auth-cookies are not specific to each user, so if Newton shares the results of these computations across users, users can avoid redundant computation.

4.6 Case Studies

It can be difficult for us to determine whether a site follows the guidelines [107], since in some cases, the tests require Newton to infer when the user logs into (or out of) a site, changes a password, or performs repeated multiple logins from the same account. We use various heuristics to infer when some of these events take place,



(a) Number of page fetches.

(b) Time.

(c) Data.

Figure 22: Page fetches, time, and data required to compute auth-cookies on Alexa top sites. In spite of our sub-optimal DNF terms generation algorithm, 90% of sites with fewer than 100 page fetches; 80% of these computations require less than two minutes to complete; and 80% require the client to download less than 20 MB of data.

including recording when a user provides input to a password box or clicks on logout links, storing encrypted passwords for later comparisons, and storing copies of cookies that were set during previous sessions. These heuristics are imperfect and will not work on all sites, but they work well enough for us to determine that many major sites are not following these best practices today. In Figure 23, the red bars show sites that violate the corresponding best practice. For each case, it was first necessary to discover which auth-cookies granted access to the website. To do so, we used Newton to first derive auth-cookies as a DNF formula, as mentioned in Section-4.2.1. In each case, we manually verified that the DNF that Newton derived was both complete and minimal. The rest of this section explores specific cases where we found a site violating a best practice, using these DNFs as the basis for our analysis. Testing for session fixation attacks requires another valid attacker session running, which is more difficult to automate.

4.6.1 Secure Against On-Path Attackers

We found that 64 out of 149 sites served at least some content over HTTPS, and yet had auth-cookies combinations that were unprotected, and would thus be vulnerable to on-path attackers (*e.g.*, at a compromised WiFi hotspot). On all 64 sites, either the HSTS policy was absent or was incorrectly implemented. Our more detailed

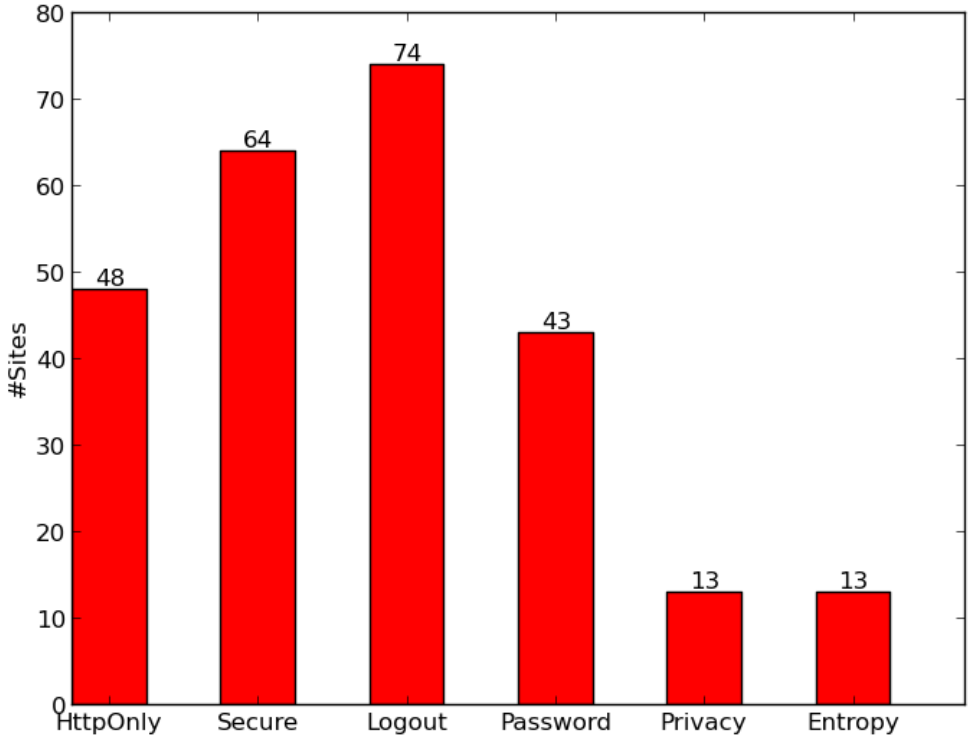


Figure 23: Results from our pilot study. "HttpOnly" indicates sites that failed to set that flag. "Secure" indicates sites that are susceptible to MITM attack. "Logout" indicates sites that did not invalidate auth-cookies post logout at server side. "Password" indicates sites that did not invalidate auth-cookies post password change. "Privacy" indicates sites that did not invalidate auth-cookies post logout at browser. "Entropy" indicates sites that did not change value of auth-cookies across sessions

evaluation provides information about HSTS policy for each site [99]. In general, when a site's contents are served over *HTTPS only* (which was the case for fourteen sites where we found vulnerabilities), not setting the `secure` flag indicates developer oversight. When a site serves content over HTTP and HTTPS, vulnerabilities may exist because web programmers have difficulty understanding how different auth-cookie combinations grant control to different parts of a website.

We present two examples of high-profile websites that have used the `secure` flag incorrectly with auth-cookies. In both cases it is interesting to note that the developer used the `secure` flag—indicating that the developer was aware of the need to secure

Table 4: Sites that acknowledged bugs discovered with Newton.

Site	Fixed?	Notes
Yahoo	✗	Even if a user always accesses Yahoo over HTTPS, search history, notes, and stock list can be accessed by attacker. Yahoo told us that due to code complexity, they require a major overhaul to fix this issue and will not be doing so currently.
Vimeo	✓	Even if a user always accesses Vimeo over HTTPS, his entire account can be taken over. Vimeo fixed this after we notified them.
Magento (~250K sites)	?	Magento developers did not specify if they are working on a fix
WooCommerce (~650K sites)	✗	WooCommerce developers blamed the vulnerability on the underlying WordPess framework.
BigCommerce (~50K sites)	✓	We commend BigCommerce developers on speedy verification and pushing a fix in the production version.
Amazon	In progress	A cross-site scripting attack can result in session hijacking. An attacker can access user's account completely. Amazon responded that they are aware of this bug and are working on a fix.

certain cookies—but still did so incorrectly, resulting in vulnerabilities such as those described in Section 4.2.

Yahoo. Newton found that Yahoo Mail had auth-cookies `Y` AND `T` AND `SSL` and Yahoo Search had auth cookies `Y` AND `T`. Although both of these services

serve content over HTTPS, only the `SSL` cookie is protected using the `secure` flag. This protects Yahoo Mail from cookie theft but leaves other services, such as search, vulnerable.

FlipKart. FlipKart is the largest eCommerce retail site in India with annual revenue of more than a billion dollars. Newton discovered that `FlipKart` has the following two auth-cookie combinations that grants access to a user's private account over HTTPS: (`VID` AND `NSID`)OR `SN` None of these auth-cookies have the `secure` flag set, rendering both combinations vulnerable to theft.

GoDaddy. Until recently, we observed that `GoDaddy` had three different auth-cookie combinations that allowed access to a user's personal account. Out of theses three, only one combination was correctly protected. We recently tested GoDaddy again and found that they have independently fixed this issue, securing all three auth-cookie combinations. This behavior shows that developers understand the threat of auth-cookie theft and are willing to fix it once it is discovered. Tools such as Newton are vital for automating this process by revealing such issues at development time through black-box testing.

Cost of implementing this recommendation: Setting the `secure` flag for auth-cookies of sites that are accessed entirely over HTTPS is easy. For sites that offer mixed content, developers must ensure that the auth-cookie combinations required for parts of the site served over HTTPS are mutually exclusive from those required for HTTP parts of the site. After that, it is necessary to set the `secure` flag for at least one auth-cookie from each auth-cookie combinations that allows access to the HTTPS parts. As we discussed in Section 4.2.3, an alternate way to tackle this problem is using HSTS, although deploying HSTS has proved to be difficult in practice.

To verify that this defense actually works in Newton, we implemented this attack and tested whether Newton could prevent it. We first verified that the Yahoo! Search's auth-cookies `Y` and `T` can be stolen by manipulating a third-party website even if Yahoo! is always accessed over HTTPS. We then harvested the user's search history using the stolen auth-cookies that we computed with Newton. Newton then forcefully sets the `secure` flag on one of them, at which point we verified that we could no longer execute the attack. (We only tested the runtime defense on the Yahoo Search site, but a similar runtime defense should be effective on other sites, as well. Similarly, if Newton detects that none of the auth-cookies in a particular combination have the `HttpOnly` flag set, it will force this flag on one of the auth-cookies to make the entire combination robust against XSS attack.) As mentioned in section-4.2.3, forcing flags on auth-cookies can break site functionality. However, Newton chooses minimal disruptions by only forcing flags on only one auth-cookie in each insecure auth-cookies set. Despite this, we are aware that such security enforcement can still break the site functionality. However, we leave the choice between security vurses functionality to user's discretion.

4.6.2 Secure Against Malicious JavaScript

One mechanism that protects against the theft of auth-cookies using XSS is to set the `HttpOnly` flag on at least one auth-cookie in each auth-cookie combination, but there are legitimate reasons why a web developer may not set this flag. For example, some JavaScript programs may need to access an auth-cookie from the browser. To secure auth-cookies, however, the programmer should always have one auth-cookie in each combination for which the `HttpOnly` flag is set to protect against XSS exploits and JavaScript from malicious third parties.

Newton found 48 out of 149 sites that serve at least part of the contents or all of the contents over HTTPS (suggesting that the content and the cookies should be

considered sensitive), yet did not set the `HttpOnly` flag to protect all auth-cookie combinations. Of these nine vulnerable sites, it appears that five of these vulnerabilities result from simple programming errors, since for those sites, *none* of the cookies had the `HttpOnly` flag set. We explore one such use case below.

Amazon. We found that Amazon AWS has the following auth-cookie combination: (`aws-at-main` AND `aws-userInfo` AND `aws-creds` AND `aws-x-main`) OR (`aws-at-main` AND `aws-userInfo` AND `aws-creds` AND `aws-ubid-main` AND `aws-session-id`). In this case, both auth-cookie combinations are protected because `aws-creds` has `HttpOnly` set to true. Yet, combinations to access a user's private account on Amazon's eCommerce site are as follows: (`at-main` AND `sess-at-main` AND `ubid-main` AND `x-main`) OR (`at-main` AND `sess-at-main` AND `ubid-main` AND `session-id`) None of these cookies have the `HttpOnly` attribute set, leaving authentication to Amazon's eCommerce site vulnerable to XSS attacks.

Cost of implementing this recommendation: A site operator can ensure that setting `HttpOnly` would not break site functionality by searching over the site's JavaScript code to determine whether any JavaScript accesses the `HttpOnly` auth-cookie. In cases where all auth-cookies are accessed by JavaScript code, the operator can secure the site by adding one more required auth-cookie to each combination that has the `HttpOnly` attribute.

4.6.3 Change Cookies Across Sessions

A capable attacker may be able to guess an auth-cookie combination if the cookie values have low entropy. Additionally, if the auth-cookie values remain unchanged across different sessions, then a single theft of auth-cookies could enable an attacker to continue authenticating to a web service as the user for an indefinite amount of time. Thus, at least one auth-cookie from each combination should change its value

across the user's login sessions, and that cookie should also have sufficient entropy to make guessing hard.

We found that 13 out of 149 websites did not change client auth-cookies across sessions. Eight of these sites also send auth-cookies in HTTP cleartext and are thus vulnerable to other attack vectors. We found Twitter and GoDaddy to be particularly vulnerable:

Twitter. `Twitter` has only one auth-cookie combination with a single auth-cookie: `auth_token`. The value of this cookie is 40 hex characters that is constant across sessions. Also, because of this, if an attacker steals this cookie, then she can login to a victim's account even if the victim is not logged in.

GoDaddy. In the case of GoDaddy, we observed that only `auth\_idp` changes in its value across sessions. All other auth-cookies retain their value across sessions, meaning that even though `auth_idp` changes, other static auth-cookies could be used to re-authenticate the user even after a session has expired.

Some web applications change auth-cookies across different sessions, but the entropy of these changes is low. For example, Facebook has two auth-cookies, `c_user` & `xs`. The `c_user` cookie is a user ID that does not change. The `xs` cookie appears to change across sessions; this cookie has 36 characters and multiple components, as follows: `119:ZHA1W3C7c7aBar:2:1406528127:6694`. Only the second part of this string is truly random; the other components are guessable, as they refer to characteristics such as the time(*e.g.*4th and 5th parts) when the user logs in.

Cost of implementing this recommendation: The wide variety of cryptographic libraries in different web languages make it possible to use cryptographic primitives to introduce more entropy into session cookies; the challenge, of course, lies in selecting good inputs to these functions that serve as reasonable sources of entropy that are

difficult to guess [120].

4.6.4 Invalidate Cookies Upon Logout

When a user logs out, at least one auth-cookie from each set used for that session should be invalidated; otherwise, an attacker could masquerade as a user simply by stealing a user's auth-cookie set. Newton found 74 out of 149 sites where past session cookies allowed an attacker access to an account where the user was logged out. Additionally, upon logout, all auth-cookies should be deleted from the user's browser; at the very least, their values should be reset. Otherwise, a sufficiently long auth-cookie can still allow the webserver to track a user's activities, even after a user logs out!

Newton found thirteen sites that do not delete their auth-cookies from a user's browser after the user has logged out, including Amazon, as described in the example below:

Amazon. After a user logs out of Amazon, both `session-id` and `ubid-main` retain their values from the logged in session. Because each of them is a 17-digit number, each of them provides enough entropy to identify each user. (`at-main` AND `sess-at-main` AND `ubid-main` AND x-main) OR (`at-main` AND `sess-at-main` AND `ubid-main` AND `session-id`).

Cost of implementing this recommendation: The difficulty of implementing this recommendation depends on the complexity of the website. Invalidating a user's auth-cookies upon logout requires deleting the state of a user's past session. The growing popularity of NoSQL databases that provide only eventual (rather than strict) consistency, however, means that implementing this feature correctly would require some care in these cases. Deleting all auth-cookies from a user's browser should be simple, though. All it requires is identifying auth-cookies and reseting or deleting

them upon user logout. A large site such as Amazon may require a careful analysis of the auth-cookie before simply deleting all cookies.

4.6.5 Let the User Control Auth-Cookies

An attacker can also steal auth-cookies by stealing a physical device; in such a case, a user should still be able to remotely invalidate his or her auth-cookies on the lost device. For this reason, we recommend that at least one auth-cookie in each combination should be derived from a user's password in some cryptographic way such as HMAC. With this rule, even if a user's password was leaked to an attacker, and if she is using it to login to victim's account, as soon as the victim realizes this and changes his password, all the auth-cookie combinations currently in use by attacker's session immediately become invalid.

On 43 out of 149 sites (including Monster, Mailchimp, Quora, OKCupid, GoDaddy, Comcast, and Amazon), we obtained access to a user's account after changing the password, using old auth-cookies that were generated in a past session which was established after logging in using an older password. (We performed our tests of stale auth-cookies approximately ten minutes after changing the user password; some of these sites may invalidate auth-cookies on a slower timescale.) This vulnerability means that if a user's auth-cookies are compromised, an attacker can retain access to the user's account *even after the user discovers a compromise and changes his password!*

Cost of implementing this recommendation: Existing websites already typically store user passwords as a cryptographic hash. This existing input could in turn be used to derive an auth-cookie that changes whenever the user's password changes.

4.7 Limitations and Future Work

Some sites have more complicated authentication mechanisms that can frustrate Newton's heuristics. We briefly discuss some of these limitations below, as well as possible ways to overcome them.

Alternative authentication mechanisms. One limitation of Newton is that it can only perform authentication tests for sites that rely on cookie-based authentication; sites that use more complex authentication are more difficult to evaluate. For example, on certain banking websites, a subsequent request also must contain request-ID from the last successful page load. These mechanisms are rare, presumably because they are not particularly user-friendly.

Large auth-cookie sets. Newton does not always enumerate the complete set of auth-cookies if the set is too large. For example, some sites such as Bank of America have many auth-cookies. Newton identified 28 of the auth-cookies on the Bank of America site but could not identify the complete set of auth-cookies in a reasonable amount of time. Although for this particular site, Newton could not completely enumerate all auth-cookies, Newton determined that the `SMSESSION` is necessary for authentication. In this case, `SMSESSION` has both the `secure` and `HttpOnly` attributes, so we were able to determine that Bank of America's auth-cookies were secure. One way to fix this issue is to use apriori algorithm for candidate generation. We are working on this and it would be released in the future version.

4.8 Summary

Web application programmers need better tools to evaluate the security of the cookies that are used to authenticate users, as well as recommendations for best practices for securing these cookies. Towards these goals, we have developed Newton,

a tool that can discover all auth-cookie combinations that permit a user to access to a particular part or sub-service (*e.g.*, Amazon Web Services vs. Amazon Shopping, Google Mail vs. YouTube) of *any* site. Our analysis of 149 popular websites revealed security vulnerabilities in auth-cookie combinations on 65 sites that could be exploited by relatively weak attackers. Many of these sites acknowledged and even fixed the bugs that Newton discovered. As web authentication continues to become more complex, tools like Newton will be increasingly valuable to users, web programmers, and website administrators. We are planning a public release of Newton. Although we have designed Newton mainly as a tool for developers and penetration testers, Newton operates as an unobtrusive Chrome extension, so ultimately even security-savvy users can use it to test the security of sites that they regularly visit. Ultimately, crowdsourced data about auth-cookie mechanisms may help both users and web developers quickly identify and eradicate vulnerabilities in auth-cookies.

CHAPTER V

PROTECTING INFORMATION IN THE CLOUD

5.1 Introduction

Web applications offer users a range of services, including email, social networking, online banking, stock trading, and subscription-based audio and video streaming services. In certain cases, corporate employees may also use Web applications to access confidential information. Developers of Web applications may use a variety of programming languages, Web servers, load balancers, caches, Web application frameworks, in-memory key-value stores, and databases—each of which may have vulnerabilities. A vulnerability in any of these components leaves the entire Web application susceptible to attacks. As a result, many recent data leaks have exploited vulnerabilities in Web applications to access private files and databases. According to a recent report, about 90% of data leaks occur at servers, and 95% of those leaks are caused by external attacks [46]. Web application vulnerabilities have led to attacks in many industries from finance [72] to large enterprises [36] to healthcare [71]. Such compromises can be devastating. For example, Sony Corporation's Web properties were compromised using SQL injection attacks during March and April 2011; attackers stole and publicized personal and credit card information [36]. The cleanup cost from this incident cost Sony $171 million, with as much as $20 billion of indirect costs [58]. Similarly, Citibank suffered a breach of 360,000 user accounts due to a vulnerability in their online banking application, which allowed one user to access another user's account by merely modifying an account number field in the URL [72].

Because most Web applications have complete access to databases, once the Web

application is compromised, an attacker can typically access any information in the database. The Open Web Application Security Project's top-10 Web vulnerabilities [104] include six server-side vulnerabilities that can result in data leaks. For example, a SQL injection attack occurs due to a Web application executing a SQL query without properly sanitizing user input. Insecure direct object reference and local file inclusion also result from poor access control checks in the Web application.

Unfortunately, building security or better access control into any given Web application framework offers limited benefits, since Web development frameworks remain in constant flux. State-of-the-art approaches scan content to block certain types of traffic: to block incoming attack traffic, organizations use Web application firewalls that scan input to a Web application to determine potential attack traffic (*e.g.*, user input that matches a regular expression for SQL injection), or data loss prevention (DLP) devices that scan outgoing data from the Web application for sensitive data patterns. Web application firewalls can only block known attack patterns so they cannot defend against new vulnerabilities or zero-day attacks. DLP systems usually have one-size-fits-all policies (*e.g.*, do not let any credit card information leave unencrypted), and are not useful for protecting Web applications that legitimately need to send and receive sensitive information.

Existing defenses check the Web application itself for vulnerabilities or perform penetration testing so as to search for flaws in the same way that attackers do. Unfortunately, these approaches cannot defend against all zero-day attacks, bugs, or vulnerabilities in future versions of the Web application. Once compromised, Web applications are vulnerable to data leaks; existing defenses such as network firewalls offer little protection for the sensitive data itself. Another possible approach to protecting against data leaks is to isolate each Web session in a virtual machine, as is done in CLAMP [109]. CLAMP provides strong guarantees against data leaks, but cannot scale to large number of users, since it requires the server to spawn a virtual

machine for each user.

Rather than focusing on creating perfectly secure Web applications or blocking all attack traffic, we posit that application compromise is inevitable and, hence, that the underlying system should *ensure that data remains secure, even if the application itself is compromised.* Towards this goal, we present SilverLine, a system that tracks the flow of each user's data between the components of a Web application. SilverLine trusts only a small portion of user-level code on the Web server that authenticates that application's users; the small size of this module could permit formal correctness verification. SilverLine places this module into a trusted realm. Thus SilverLine entrusts the attestation provided by authentication module for a specific user's session and permits outgoing data on client Web sessions only if the user is authorized to retrieve the sensitive data. Even if an attacker gains access to sensitive data by compromising a Web application or other user-level processes, SilverLine prevents that data from being leaked, because the attacker's Web session cannot prove that it is authorized to access the data.

SilverLine associates a *taint* with each piece of data that the Web application accesses, such as a database record or a file. To protect data independently of the applications that access it, and to track information flow, SilverLine instruments the operating system kernel of the Web application server with a trusted module that associates a taint with each process and file. The SilverLine kernel module propagates these taints as files and data objects are accessed or modified; the kernel propagates these taints regardless of whether the data is transferred within a single host or to another host on the network. When a Web application reads a user's data, SilverLine propagates the data's taints through each Web application component, and associates them with network packets that the Web application generates. SilverLine's declassifier module then checks the taints on the outbound flow with the set of taints that the user is authorized to see. If they match, the traffic flow is allowed; otherwise,

the flow is blocked. Although we have implemented SilverLine's declassifier as a standalone control application, it decides whether to allows flows based on taints that the flow carries and could thus be implemented as an application as a *Software Defined Networking (SDN)* control application in an existing OpenFlow [102] controller (*e.g.*, Pox [115], Floodlight [62]).

SilverLine separates Web application security from the security of its data, ensuring that a user's data is secure even if the Web applications that access the data are compromised. Because SilverLine performs information-flow tracking at the operating-system level, it can defend against attacks on any application or service that touches the data, not just attacks against the Web application. Because it does not require virtual machines to isolate data from different users, it reduces throughput by only 20–30% compared to a native system without any data leak prevention, which is significantly less performance overhead than existing approaches (the state of the art, CLAMP, incurs a 42% reduction in throughput).

This paper presents the following new contributions: First, we present SilverLine, a system that prevents data leaks in vulnerable Web applications that is independent of any particular Web application and requires only minimal changes to existing Web applications. As part of designing and implementing SilverLine, we develop a new mechanism for associating taints with Web application users. We also develop a framework that makes it easy for administrators to configure SilverLine policies. To show that SilverLine is practical and scalable, we port a widely used real-life application that is used in more than 12,700 online stores, OSCommerce [103], to run on SilverLine. We show that SilverLine imposes less overhead than existing data leak prevention systems and scales to many more users and requests than these existing systems.

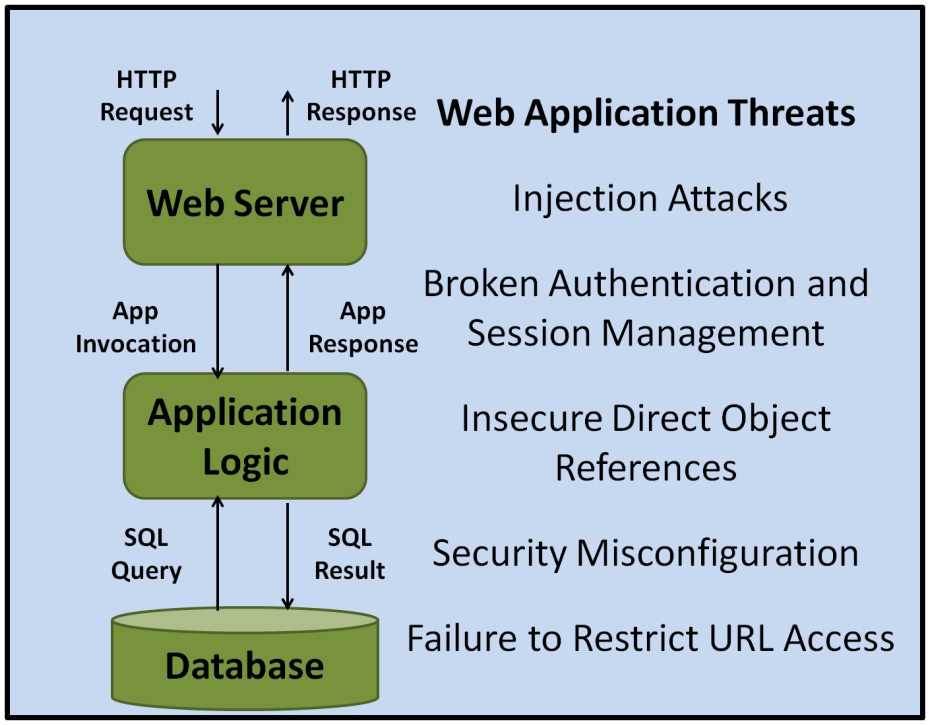


Figure 24: Web server stack and threats that SilverLine defends.

5.2 Design Goals

SilverLine protects multi-user Web applications where a single user logs in over a session. Figure 24 shows a typical Web application deployment scenario and the set of threats that we address in this paper. We aim to defend against data leaks in the case where an attacker can compromise the Web server, user libraries, or the Web application code itself. An administrator for the Web application must identify the sensitive data and apply the appropriate initial set of taints; we believe that this approach is reasonable because the sensitive data accessible through Web applications is easy to identify and protect in advance. We aim to achieve the following goals:

Security: Decouple data protection from the application SilverLine prevents data leaks, but it does not protect the Web application from exploits. By identifying private information when the database schema is designed, SilverLine can prevent data leaks independent of specific queries or application compromises. SilverLine

may still be vulnerable to certain types of data leaks, such as covert channels, but it can defend against any data leak attack that results from the application having complete access to the back-end database (a common scenario for Web applications).

Performance: Minimize overhead We aim to keep the performance overhead of deploying SilverLine as low as possible, by reducing the overhead associated with tracking data in Web sessions. In contrast to existing systems such as CLAMP, SilverLine does not require spawning a new virtual machine for each new Web session, which improves both scalability and performance.

Deployment: Minimize changes to existing applications Due to the large deployed code base for existing applications, our goal was to allow SilverLine to be deployed for existing Web applications, without requiring significant rewrites of application code. SilverLine only requires applications to have a secure login module, that associates Web sessions with particular users. Other than the introduction of this secure login module, SilverLine does not require any other changes.

"Non goals" and vulnerabilities SilverLine also has several "non-goals". SilverLine only defends against attacks on user-space applications (*e.g.*, the Web application), not on the operating system that hosts the application; kernel code injection could disable SilverLine's information-flow tracking. SilverLine does not protect against malicious software on the database server that might directly exfiltrate sensitive data without passing through SilverLine's data firewall[1]. SilverLine also does not consider threats posed by insiders or others who have physical access to machines that hold sensitive data. Because SilverLine protects data rather than identifying

[1]This is a reasonable assumption: we expect that the database server is firewalled from being directly accessed through the public Internet. Furthermore, if the database server software was malicious, it could return sensitive database records even for non-sensitive queries.

malicious input, it cannot defend against malicious attacks that modify data. Because SilverLine performs no additional access control on data flowing *into* the Web application, an attacker might still be able to make the Web application execute harmful commands (*e.g.*, a "DROP TABLE *" query using SQL injection); we believe that these attacks are less serious than data leaks, because periodic database and transaction log backups make it possible to restore the database at a five-minute granularity [33].

5.3 Design

We first present an overview of the SilverLine design and describe how the system integrates with a current three-tier Web architecture. We then explain how taints are managed and propagated, and how SilverLine applies information-flow control to prevent unauthorized data access.

5.3.1 Overview

Restricting information access requires knowledge of which rows a database query will select, which can be difficult in practice and also restrict the Web application developer. Many existing systems do not allow one user to query another user's data [109]. They also do not allow queries from an anonymous context, thereby making aggregate queries difficult or impossible. In SilverLine, application code and database queries can access all data (thus making aggregate queries easier to perform), but exfiltration is restricted based on the user who attempts to access the data, which provides significantly more flexibility.

SilverLine has the following trusted components: the *database proxy*, which applies taints to retrieved data; the *user authentication* component; and a *network-wide information flow control system*, which propagates taints. SilverLine taints database records by augmenting a user's database records with an additional "users_taint" table, and using a database proxy that sits behind the database server and the Web

application to transparently apply these taints before data is returned to the Web application. We provide a brief overview of each module below.

Database proxy Requests sent to or received from a sensitive database pass through a trusted *database proxy*, which can identify database requests for sensitive data, and appropriately taint the responses before they are sent back to the untrusted Web application. This proxy allows the Web application to function exactly as it does without SilverLine—it fetches sensitive data from the database, processes it, and returns it to the logged-in user's session. This approach makes it easier to deploy SilverLine with existing Web applications.

Authentication module A legitimate user of the Web application who accesses the landing page may be prompted to login using code served directly by the trusted *authentication module*. The authentication module uses a separate, trusted database to authenticate the user and sets policies for the *application* firewall; these policies specify the types of sensitive data that the user is allowed to see. The authentication module then forwards the user's request and the token indicating successful authentication to the core Web application. As in similar systems, the authentication module must be trusted. *The authentication module is the only part of existing applications that needs to be rewritten.*

Information flow monitor The Web application runs on a *network-wide information-flow control system*. SilverLine uses a kernel-level module to track the flow of information from tainted sensitive data to applications. Much like taint-tracking, it can propagate taints between files and processes, but can also propagate taints across a network—a key requirement for distributed Web applications. After the Web application processes sensitive data, the responses it generates are forwarded through a simple programmable firewall component, the *declassifier*. The declassifier relies on

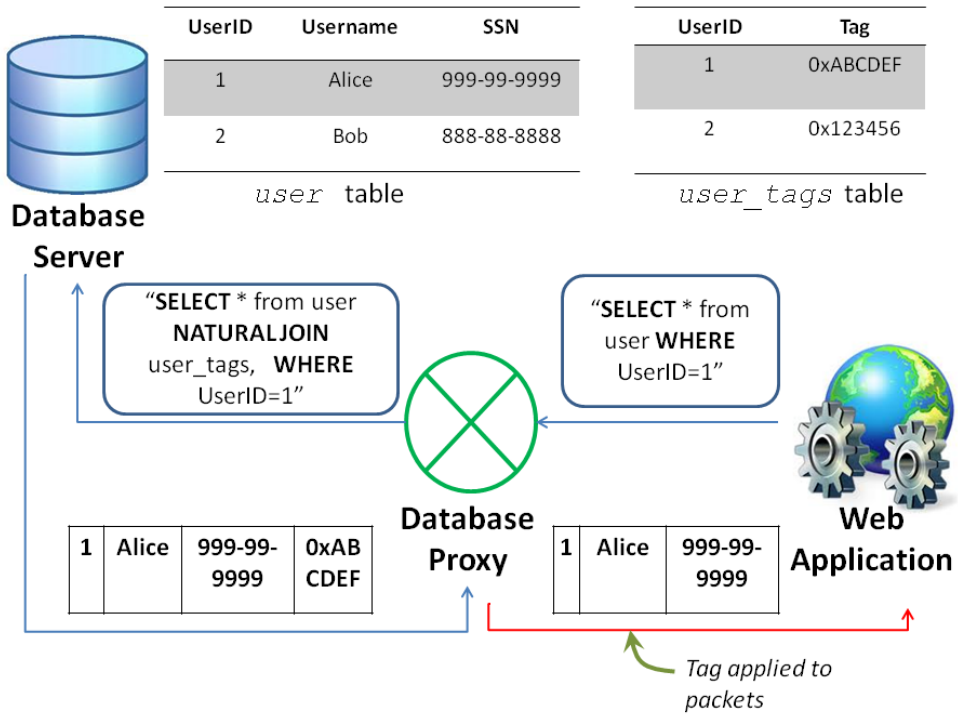


Figure 25: Rewriting retrieves taints associated with the result.

the authentication module to compare any taints on outgoing network flows with the authorization associated with the client session: if the user is authorized to read data with a certain taint, the declassifier forwards the data; otherwise it drops it. Thus, even if the attacker compromises the Web application, the attacker's session cannot read the data.

5.3.2 SilverLine, Step-by-Step

Step #1: Administrator tags sensitive data SilverLine relies on the database administrator (DBA) to mark sensitive information in the database tables. It also requires the DBA to specify primary, candidate, and foreign keys, as well as referential integrity constraints between them. For example, a sensitive table may contain the primary key *User-ID* and foreign key *Transact-ID*. This foreign key might be used in another table, `Transaction`, as a primary key. SilverLine creates a unique taint for each user and stores taints for all users in a separate table `User-Taints` with *User-ID*

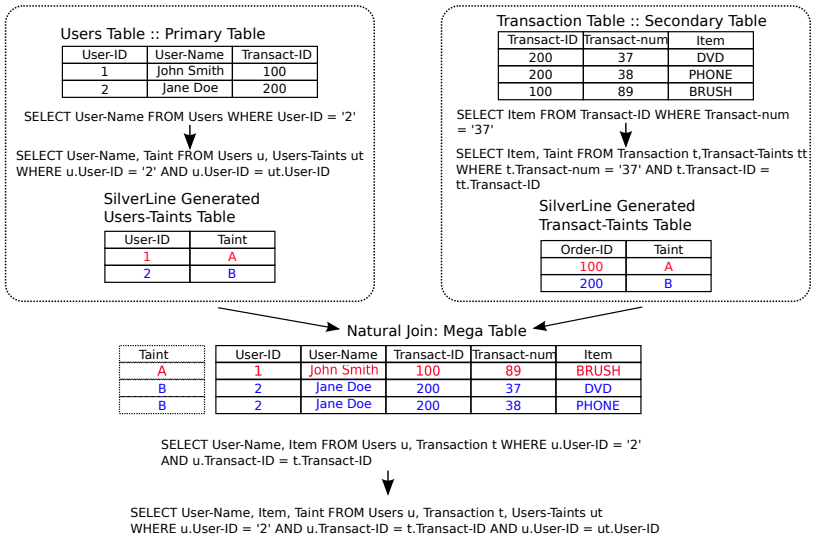


Figure 26: Example of how SilverLine stores taints and associates them with records belonging to each user.

as the primary key. Storing taints for data in a separate table prevents the need for drastic changes to application logic or the database proxy if the database schema changes, although it incurs some additional overhead when joins occur. All tables with private data might not have a *User-ID* as primary key. Querying one of these tables would thus require multiple joins between tables, first to find the *User-ID* and then to locate the associated taints, which would slow performance. To reduce performance overhead in these cases, SilverLine generates additional taint-storage tables for each foreign key. For example, Figure 26 shows a taint-storage table, `Transact-Taints`, corresponding to the `Transaction` table.

Step #2: The authentication module associates a user with a session The application's authentication module performs user logins by comparing usernames and passwords input by clients with the hashed password database of the Web application. New client requests to the Web application first encounter a login Web page hosted by the authentication module. After users enter their username and password, the login module accesses the `User-Authentication` table to authenticate the user. If the login succeeds, the module performs two actions. First, it creates a new Web cookie,

returns the cookie to the client, sets the cookie in a `User-Sessions` table indexed by the user ID, and redirects the user to the main Web application. The Web application can now use the `User-Sessions` table to verify that the client is authenticated. Second, it creates a new record in a `Connection-Capabilities` table that specifies the set of sensitive taints data may carry over the user's connection; a connection is identified using the 5-tuple of source IP address, source port, destination IP address, destination port, and protocol. The `Connection-Capabilities` table is set by querying the `User-Taints` table. Web applications typically need to distinguish between the users who may have access to particular data, which allows SilverLine so track taints at a per-user, per-process level. These coarse-grained taints incur less performance overhead than other information-flow control systems.

The authentication module is the only component in existing applications that involves refactoring within an existing Web application. First, the module must be trusted and hardened against compromise; otherwise, an attacker could masquerade against any user to leak data. Second, we ensure that the authentication module is the only module that can access a user's password; otherwise, a compromised Web application could modify users' passwords to gain access to sensitive data. The `User-Authentication` table is also stored in the same database used by the Web application, but SilverLine relies on the trusted database proxy to block direct modifications to this table from the Web application.

SilverLine applies taints at the granularity of processes, which reflects a particular design choice in the tradeoff between performance and security. Tainting each thread independently would likely incur less performance overhead, but such tainting would have to take place in user space, where malicious processes could be more capable of interfering with or compromising the tainting process. Tainting at the process level incurs more overhead by comparison, but allows all tainting to take place in the (trusted) kernel.

Step #3: The database proxy rewrites user queries to retrieve taints associated with a result set SilverLine's database proxy parses and rewrites queries or responses to inter-operate with the SilverLine back-end, as shown in Figure 26. Database proxies such as MySQL proxy [95] are becoming increasingly popular in the industry for database activity monitoring, query logging, and for detecting queries that may damage the database.

The *User-ID* field in `User-Taints` refers to the primary key of `Users`. When a SELECT query is issued to retrieve a user's data, the database proxy parses the SQL to identify whether the query attempts to retrieve sensitive data (based on policies that the administrator specified in the schema). The proxy then modifies the query to execute a join with the `User-Taints` table; the associated taints are then applied to the result set. We found that about 85% of OSCommerce's queries fall into simple SELECT queries that can be rewritten in this fashion. Still, Web applications sometimes perform complex join queries that may be harder to rewrite.

Figure 26 shows how SilverLine stores taints and associates them with users for an example database schema involving users and their associated transactions. The `Users` table is a primary table that associates *User-ID* with *Transaction-ID* and can be directly joined with the `User-Taints` table to retrieve taints associated with each user. When a query is executed against `Users` the SilverLine database proxy finds taints associated with each user ID by performing a simple join between the `Users` and `User-Taints` tables.

We that of the 50 tables in the OSCommerce Web application, only 15 required taints; nine of those tables could be associated with `User-Taints` table since they shared same primary key. Out of the remaining six tables, five of them shared same primary key and hence a taint-storage table with that foreign key allowed to extract the taint. In the end, we needed to create three taint-storage tables in order to extract taints by doing just one join.

To retrieve taints for queries that perform joins, SilverLine first discovers all of the sensitive tables associated with that query and determines the taint-storage table with the correct primary key. The database proxy then modifies the original query to add the taint-storage table in the join, from which it selects the taint column.

Step #4: The database proxy associates taints with the response When the Web application requests records for a user, the database proxy associates the user's taint with the outbound connection between the database server and the Web application, using a trusted, shared cache that trusted SilverLine components can all access. Because the Web application runs on an OS that includes the SilverLine kernel module, the host can pick up the taint on the incoming data and associate the taint with the Web application process that reads the data.

Step #5: The server's information flow control system tracks reads and writes associated with the tainted record Once taints are associated with any resource (typically a file or a process), the kernel monitors any system calls that transfer information from the resource, and propagates the taints to the destination of the information flow. If a process reads a tainted file, the process acquires the taints of the file. If a tainted process writes to a file, the file acquires the taints of the process. Finally, if a tainted process sends data over the network to another process, the receiving process acquires the taints of the sending process.

To propagate taints between processes on a single machine, the kernel intercepts each system call and checks whether the sender resource carries any taints. If it does, it adds all of the sender's taints to the receiving resource's set of taints. To propagate taints across the network, the sender's taint-tracking module creates an entry in the `Connection-Taints` table that maps the 5-tuple for the flow with the current set of taints of the sending process. This out-of-band taint transfer keeps implementation simple and avoids intrusive changes to the underlying system. When a host receives

the first packet of a flow, it queries the `Connection-Taints` table and applies the set of taints to the receiving process. A sender can use the Type of Service (TOS) field in the IPv4 header of the outgoing packet to indicate additional taints.

Step #6: The declassifier determines whether a user can access data. The *declassifier* determines whether a particular flow should be permitted to go to a client Web session. When the server sends the response back to the client for the first time, the declassifier verifies that the connection has sufficient privileges to carry taints associated with it by querying `Connection-Capabilities`. Once the declassifier decides that the client should have access to the data, it will not check for violations until the sending process acquires new taints (as indicated by TOS field on incoming packets). If the data in the flow should not be allowed, the declassifier terminates the connection and sends a signal to the server to kill the associated processes. We have implemented the declassifier as a standalone application that is on the path between the client and server, but in the future it could be implemented as an off-path network control application using an SDN controller such as Floodlight [62] or Pox [115]. Flow table entries permitting flows could be installed in the switches using micro-flow entries that match on the TOS field and other flow tuples; a change in the TOS bits would result in a "miss" in the switch flow table, causing redirection of the traffic flow to the declassifier, which could subsequently re-evaluate whether the corresponding traffic should continue to be forwarded.

5.4 Implementation

This section describes the implementation of SilverLine components; our integration of SilverLine with OSCommerce, a popular Web commerce application; and the process by which an administrator configures SilverLine.

5.4.1 SilverLine Components

We describe the implementation of SilverLine's *trusted components*: the authentication module, the database proxy, and the information flow control framework.

5.4.1.1 Authentication module

The login process of the Web application has two components. The request handler, executes on the Web server, as before. However, it only handles the incoming login requests. The actual authentication is performed by another process, which runs on a secure server. When the request handler receives a login request from a user, it invokes the RPC of the authentication process with username, password, and the five-tuple of the login request. This process then checks whether the username and password are correct. If it does, it retrieves the taints associated with that user and puts them in the `Connection-Capabilities` with the associated five-tuple and returns a success code to request handler. The request handler then generates a session for that user, creates a cookie, and returns the cookie to the user. **We implemented these changes in less than 60 lines in the OSCommerce source code.**

5.4.1.2 Database proxy

We wrote a 350-line Lua script that acts as a MySQL database proxy, which intercepts queries coming from web application and decides if any of them will potentially retrieve sensitive information. If the proxy identifies a query that accesses sensitive information, the proxy then discovers the taint associated with that information and applies it to the result of the query.

Taint Storage As mentioned in the design section, we create new taint-storage tables for each primary key for tables that contain sensitive data. If a taint-storage table shares its primary key with a bunch of database tables then it can be used to retrieve taints associated with any data accessed from those tables. Adding a new

Table 5: Types of system calls that are monitored by the labeler, with examples of each system call type.

Syscall Type	**Example syscalls**
Inter-process Communication	send(2), shmat(2), msgsnd(2), kill(2)
File/device operations	read(2), unlink(2), mknod(2)
Process creation	fork(2), execve(2), clone(2)
Memory operations	mmap(2), mprotect(2)
Kernel configuration	sysctl(2), init_module

primary key to any of these tables requires adding a taint to the corresponding taint storage table. The database proxy monitors inserts in these tables and creates a new query that inserts a taint along with the primary key from the original insert.

Taint Retrieval When the database proxy receives a database query, the proxy script runs a query parser. This parser analyzes the query and returns the query that will also retrieve the associated taints for that data, which is a join between the original table and the taint-storage table. It also returns a regular expression for the original query (to match similar future queries with different parameters) and a regular expression for the taint-retrieval query. The database proxy stores both of these queries in a global table for future queries. Any future queries that match the regular expression need not be passed to query parser, since the associated taints for those queries have already been retrieved.

5.4.1.3 Information flow control

Host-level information flow We implemented host-level information flow control as a kernel module on Linux, using the Linux Security Modules framework [136] in the Linux kernel. LSM is a framework within the Linux kernel that allows various security models to be added-on without changing core kernel code. LSM provides hooks within system call handlers that can be implemented by a security module; thus, a third-party module can implement mandatory access control for a system call (*e.g.*, `read(2)`) without changing the core implementation of the system call handler

(*e.g.*, `sys_read`). Using LSM hooks, the labeler intercepts all system calls that transfer information between host resources. We have implemented hooks to track information flow for the system calls in Table 5.

We expect that the module could be bundled with the OS distribution for the machines running the Web application or installed when the system is in a known clean state. On boot, the module is loaded shortly after `init` during the boot sequence. The hooks for the system calls are 1,500 lines of C code, and the logic for the module is another 6,500 lines, so the total module implementation is about approximately 8,000 lines of code.

Information flow across components SilverLine use trusted databases to communicate information from different components—from the login module to the Web application, from the kernel module to the declassifier, etc. For performance reasons, and to disassociate ourselves from the Web application's own database, we use a single centralized high-performance key-value store for storing the `Connection-Capabilities`, `User-Sessions`, `Taint-Policy`.

We choose the Redis [116] key-value store for our implementation. Redis supports about 110,000 `SET`s per second, about 81,000 `GET`s per second [117], outperforming many relational databases (*e.g.*, MySQL) and key-value stores (*e.g.*, memcached). Redis supports only string keys, but values can be of any type. Redis store maintains three tables. The first table contains tag capability information. The second table contains the user information. The third table maintains which tags are *allowed* to be associated with the data leaving the network on each TCP/IP connection. These three tables are *protected* and can only be updated by trusted components such as database proxy. This table is filled by trusted authentication module after verifying the user's credentials. The last table contain information about which tags are *acquired* by each connection. This table is updated by both database proxy and the Web application

server. To protect untrusted user space code, on the Web application server, from updating entries in any Redis table, the end-host component of SilverLine does not allow any user space code to connect to the Redis server.

5.4.2 Configuring SilverLine

All SilverLine configuration occurs at setup time. A key feature is that the administrator who configures SilverLine does not have to understand the logic for the applications that run in the environment. The administrator does need to identify sensitive data for each user; in particular, the administrator must identify (1) the primary keys for tables, (2) groupings of tables with the same primary keys, (3) relationships between the foreign keys across tables and (4) the table from each group that shares a specific primary key for INSERT statements that would result in a new primary key for that group.

When porting OSCommerce to SilverLine and inspecting other Web applications, we noticed that administrators typically already intuitively identify many of these parameters. Once the administrator specifies these four parameters, SilverLine creates taint-storage tables for each group of tables and automatically maintains them whenever a row is inserted or deleted in that group.

5.5 Performance Evaluation

We now explore SilverLine's performance and compare it to existing state-of-the-art systems with similar goals, such as CLAMP. SilverLine's main performance overhead results from retrieving taints from the database server and propagating those taints to processes corresponding to that user on the Web server. This tracking involves executing the query, associating the taints with a connection in the central monitoring database, and performing taint tracking within a host using system call hooks. Table 6 summarizes the main results from the evaluation.

Table 6: Summary of performance evaluation.

For small files, the latency overhead constitutes about 7%; for large files, it is about 1%.	§ 5.5.2
SilverLine can support on the order of 1,000 simultaneous users with realistic response-time bounds.	§ 5.5.3
SilverLine's overhead on the throughput of login requests is about 20%; the overhead on response time is about 29%.	§ 5.5.3

5.5.1 Experiment Setup

To measure SilverLine's performance overhead, we ran various experiments with and without SilverLine. Without SilverLine, we ran the experiments on the native LAMP stack. With SilverLine, we enabled the authentication module, taint tracking, and a MySQL database proxy to fetch and apply taints.

Our testbed consisted of four Emulab nodes connected in a setup as shown in Figure 27. Each node had 64-bit 2.40 GHz Intel Quad Core Xeon E5530 with 12 GB RAM. Each link is point-to-point with 1 Gbps capacity and roughly 0.05 ms latency. We ran evaluation scripts on the client node. The server runs the OSCommerce web application with the taint-tracking kernel module. The database node runs a MySQL server with the SilverLine MySQL database proxy. The "SilverLine Monitor" node hosts the trusted databases that perform information-flow tracking across nodes. The monitor and database nodes contain only modules that are completely within the trusted domain. The Web server node, which runs the Web application that also has the (modified) authentication module, has both trusted components (*i.e.*, the authentication module) and untrusted components (*i.e.*, the Web application itself).

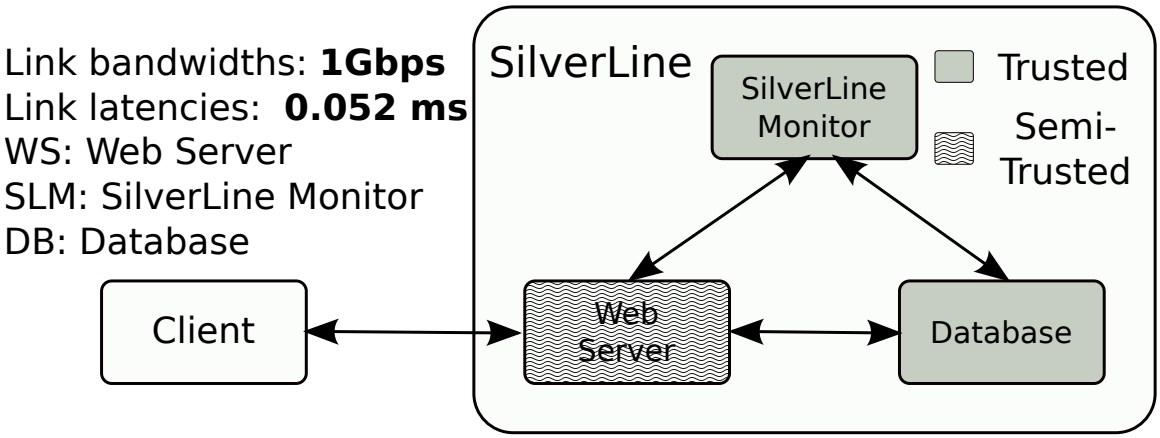


Figure 27: SilverLine: Evaluation setup.

We used the Apache Web server on the server with a prefork `mpm` module to allow only one client connection per child process. The server spawns a new child process per client connection to avoid taint diffusion across different client sessions, since tainting is performed on a per-process granularity. To prevent multiple requests from the same client being served by separate child processes, we also enabled the Keep-Alive option. Apache can spawns about 32 child processes per second. Under higher request loads, the default Apache server was unable to fork enough children quickly enough, so we modified the spawning algorithm of the Apache `mpm` module. (It is also possible to use other Web servers that can fork more child processes to satisfy this constraint.) We configured frequently updated tables (*e.g.*, sessions, shopping cart contents) using MySQL's `innodb` storage engine to take advantage of row-level locking.

5.5.2 Latency

The ultimate measure of Web application performance is perceived user experience, but it is extremely difficult to measure this metric. As a proxy for this metric, we measure the latency experienced by the end user due to SilverLine. In Section 5.5.2.1, we perform macrobenchmarks to measure overall latency increases incurred when serving both static files as well as requests that read or modify the back-end databases.

In Section 5.5.2.2, we perform microbenchmarks to understand the contribution of each SilverLine module to latency.

5.5.2.1 Macrobenchmarks

We measure performance of various operations when the web server is not loaded with multiple requests. We measure all the times from the client machine. A single type of operation such as "Login" may consist of more than one web request; therefore, we start measuring the time when the first HTTP request is issued and stop measuring it when all the requests in that operation have responded. For each operation, we perform 50 trials, and report average and standard deviation of latency. Figure 28 shows these results. In all of our experiments below, the link latency between the client machine and the Web server is 0.05 ms, which is much smaller than it would be in reality on the wide-area Internet. We chose a small link latency to allow us to provide an upper bound on the latency overhead that SilverLine induces. In realistic settings, the percentage overhead that SilverLine induces will be an even smaller fraction of overall latency.

We replicated the same test cases as CLAMP to allow us to compare the two approaches. In all test cases, however, *CLAMP omitted the time to fork a new virtual machine* and assumed that virtual machines are pre-forked, which makes their performance numbers appear significantly better than they might be in practice, when new sessions would require forking new virtual machines. In contrast, *SilverLine's end-to-end performance measurements also include time taken to spawn a new process that serves that client.* Additionally, CLAMP requires each session to be served over SSL to identify each session. We do not have such a constraint with our prototype and can serve requests with or without SSL transport. Thus, we chose to measure performance without SSL.

In the first set of measurements, we fetched small (8 KB) and large (3 MB) static

files that did not involve database operations. To measure the performance overhead of SilverLine, we added single taint to each file. This static file transfer measures the overhead of taint tracking on a single host. *For small files, the latency overhead constitutes about 7%; for large files, it is about 1%.* File size does not affect the latency overhead associated with taint tracking, so for large files, that cost is amortized over the total file fetch time. Typically, a Web page is hundreds of kilobytes or more, thus making the percentage overhead of this fixed cost smaller. In contrast, CLAMP's latency overhead increases as file sizes increase.

To test the login overhead, we measured the time to redirect to the monitor for secure login, retrieve taint from the database, and apply that taint to the user session. Overall, this function adds about 13 ms (30% in the worst case).

5.5.2.2 Microbenchmarks

To analyze how much each SilverLine component contributes to the latency overhead, we simulated a normal user operation and measured the time taken by each component. In this case, a user logs in, checks her cart that already has a few items added, visits a few more product pages and then logs out of the system. In terms of SilverLine components, the overhead involves redirecting to the monitor for secure login, retrieving the user taint, applying that taint to the user session and finally making a decision whether to let the information to be passed to the user browser. We found that the login process contributes about 8% of the overall latency overhead. The database proxy contributes 61% of the total latency overhead. Propagating taints to the Web server via monitor contributes 21%. Finally, taint tracking within the Web server and declassification checks contribute 10%.

Finally, we measured the overhead of performing database operations after the user has already logged in. This measures the overhead experienced due to addition of MySQL proxy server that retrieves and propagates taints. The operation involved

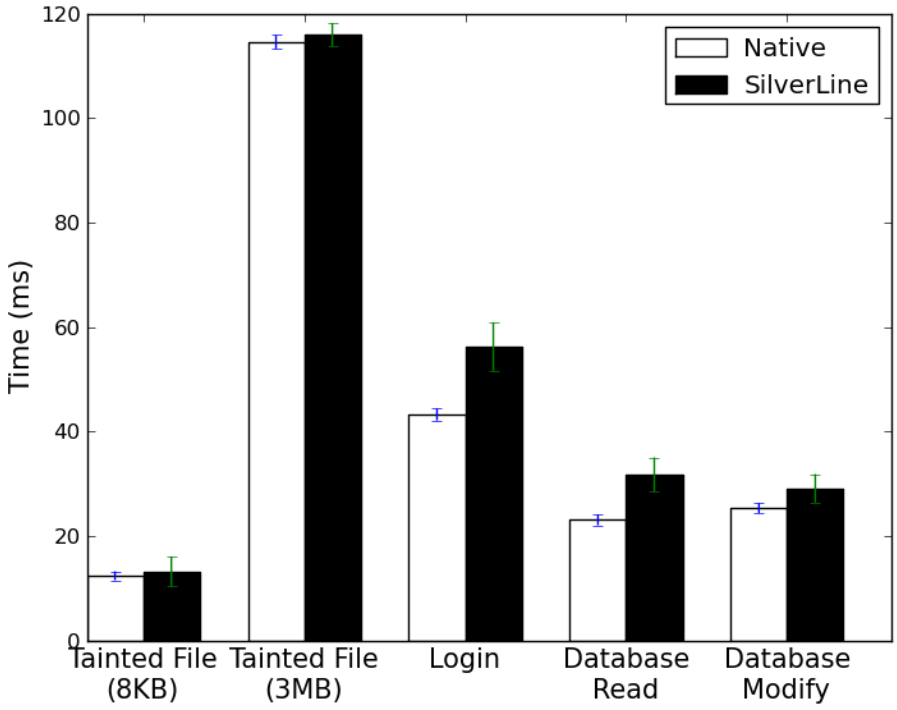


Figure 28: **Macrobenchmark Latency.** Comparison between native LAMP installation and SilverLine setup to complete various requests. Smaller is better.

about 18 SELECT queries introduces an additional 8 ms overhead; the database update includes 20 INSERT or UPDATE queries and introduces a latency overhead of 4 ms.

5.5.3 Scalability

Since SilverLine does not consume much extra memory per process, the upper bound on simultaneous user connections (without considering the response time) is equal to maximum number of server processes that Apache Web server can spawn. Often, the Web server has a hardcoded limit of 20,000 server processes in total. We are able to support on the order of 1,000 simultaneous users with realistic response-time bounds.

To determine whether SilverLine can be deployed without affecting the scalability of the Web application, we generated parallel requests at the Web application and measured the resulting throughput as the number of sessions increased. Because most

of the standard benchmarks do not simulate an actual user's navigation for a specific Web application, we created our own set of tests using a Python and Curl framework. These testing scripts spawn multiple threads and perform various user actions such as logging in, browsing, adding items to shopping cart, checking out, and logging out. In general, we found that the throughput involving concurrent login requests only is reduced by 20%, while tests simulating multiple realistic user sessions had 30% lower performance. Most of the throughput reduction comes from the SilverLine database proxy that rewrites and retrieves taints. We found that even with this performance hit, SilverLine was able to serve several hundred requests per second. We also note that latency (not throughput) is increasingly the factor that many Web applications face, so even this throughput reduction may be manageable in many cases.

Login Time vs. Number of Users We measured how login redirection affects the number of concurrent login requests that the Web server can serve. We issued 5,000 login requests. For each trial, we increase load in terms of the number of concurrent requests per second. We stopped increasing the concurrent load when the average response time for requests exceeds two seconds. We measured the maximum concurrent load per second that can be served within two seconds as the login throughput of the system. On the native server, the Web application could support 349 login requests per second. After enabling SilverLine, the application was able to support 280 requests, an effective throughput overhead of about 20%.

Response Time vs. Number of Users Usually on e-commerce sites such as those where SilverLine might be deployed, different users are in different modes of interaction: some users are logging in while others are performing a transaction. To simulate this environment, we wrote a script that simulates a complete user session with logging in, adding a random item to a shopping cart, checking out, and logging out. These four actions typically generate more than ten HTTP requests. To see the

performance of the site under such load, we ran the script for 5,000 users, increasing the number of concurrent sessions per second and observing performance. We stopped increasing the concurrent load when the average response time for each session exceeds 20 seconds. We measured the maximum concurrent load per second that can be served within 20 seconds as the user session throughput of the system. On the native server, we see an average 187 requests served per second; with SilverLine enabled, we measure 134 requests per second, an effective throughput cost of about 29%. We also measured how many concurrent users can be logged in simultaneously, and the effect of simultaneous user requests on latency. In this case, SilverLine could support approximately 987 concurrent users, and the 95th percentile of these requests were served in less than two seconds.

5.6 Limitations and Future Work

Misconfiguration In Section 5.4.2, we discussed how a new Web application can be configured to use SilverLine. From the database schemas that we inspected of the target category of applications, we found that finding the required configuration information is reasonably intuitive. Nevertheless, in case of a misconfiguration such as a table with sensitive information being misclassified into a into a non-sensitive group, data leaks can still occur (although the leaks would be limited to data in that table). We believe that misconfiguration could be detected early, with test cases where a particular user's session leaks information about another user. Because our taint storage granularity applies to an entire row of each table, a single test case per table should sufficient to detect misconfiguration. With such a sanity test mechanism in place, the administrator can iteratively refine the configuration of SilverLine.

False positives and false negatives Any taint tracking system would be required to avoid (1) taint explosion, where information gets associated with way more taints than the correct number, resulting in false positives; and (2) a failure to associate

the required taint with the sensitive information, resulting in false negative. False positives hampers the otherwise correct application functionality and is a trouble for administrators, while false negatives are silent and defeat the data-leak prevention. Because we are focusing on Web applications that have strong isolation between users' data and sessions, we do not expect a user session to be tainted with more than one taint in normal operation. Still, some database queries can run across an entire sensitive table and aggregate results, which makes SilverLine most suitable for applications where a single user is accessing data that is not aggregated across multiple users (*e.g.*, banking, e-commerce). An application that runs aggregate queries with a WHERE clause are currently not supported. Such applications would require functionality change. We hope that we will identify such queries quickly as they break that part of application due to exfiltration prevention. Some other techniques that we have not investigated in details could be applying differential privacy measurements to such queries and letting the resulting answer pass through only if it is below a threshold.

In the absence of covert channels, false negatives are mainly a combination failure to identify sensitive information and failure to track the sensitive information. The failure to identify sensitive information can be fixed with a few test cases. Proving that taints are tracked correctly is more difficult job. In SilverLine, we track taints at the process level and we only do so at specific points which are system calls. Ultimately, it may be possible to create a formal model for information flow for each system call and prove the accuracy of taint tracking.

Protection against data modification We focus on preventing information leaks. We believe that protection against acts where somebody deletes a row or drops a table can be mitigated with appropriate backups. Even then, of course, an attacker could modify another user's credentials and impersonate his identity. Although we do not address this problem, we believe that SilverLine could be modified to defend against

such an attack by monitoring INSERT queries and verifying that a user session has the taint corresponding to the row that it is trying to modify. We describe this defense in our discussion of retrieving sensitive data (Section 5.3.2).

Partial Deployment Administrators may wish to deploy SilverLine for only a subset of users. Because SilverLine does not modify the original application or the original application's database schema, it can support such a partial deployment. In that case, the administrator must configure the list of users for whom the system would be enabled. The login module can periodically check whether the user belongs to this list. All of the other mechanisms such as taint retrieval and association, taint tracking, and declassification would be enabled only if that check is successful.

Integration with social networking applications Social networking applications are not suitable for SilverLine, since they mainly thrive on information diffusion. Many Web applications now include some aspect of social networking such as "Share what I just did on Facebook". This type of integration can be fixed by reclassifying sensitive information tables or by maintaining shadow database tables which are non-sensitive. Certain information might be explicitly declassified before it is shared with public users.

Integration with existing SDN controllers. The current implementation of the SilverLine declassifier is a standalone application that resides on the path between the client and server. In an OpenFlow (or SDN)-enabled network, however, such an application could be implemented as an off-path network control application running at an SDN controller such as Floodlight [62] or Pox [115]. Moving these functions into a general SDN control application could ultimately allow information-flow policies to be expressed in higher-level languages such as Pyretic [93].

5.7 Summary

We presented the design, implementation, and evaluation of SilverLine, a system that prevents data leaks that might otherwise result from Web applications. SilverLine represents a different point in design space for Web application security systems: Rather than focusing on application modifications or data access control, SilverLine prevents exfiltration of sensitive data, even if the Web application itself is compromised. SilverLine uses process-level taint tracking to isolate code executing on behalf of various users. By applying taint-level protection to the data, rather than to the application or execution environment, SilverLine can achieve strong guarantees, while scaling better and incurring less performance overhead than existing state-of-the-art systems. It also simplifies runtime checking for data leaks, since it does not rely on specifying complex data-access policies. Our prototype of SilverLine indicates that augmenting a wide variety of Web applications with SilverLine requires only modifications to the login process, not to any other part of the Web application.

CHAPTER VI

CONCLUSION

In this dissertation, we have presented multiple security tools that focus on automating the translation between "security semantics" and "security syntax" to reduce errors in the translation process and also to make it faster. These tools mainly protect attacks against web services that are targeted at various vantage points in the World Wide Web. In each case, the tools take advantage of the specific properties present in the sensitive information that is protected to make the translation process faster. In the following section, we first summarize the contributions about how each chapter automates the translation from semantics to syntax and then discuss possible future directions.

6.1 Summary of Contributions

In Chapter 3, we presented Appu, a browser extension that helps users to keep track of their personal information spread across various web services. We first developed our own language, "FPI" to automatically detect users' personal information. Using this personal information, Appu also monitors for the further spread of this information to see if it is present on other webpages. Over time, with this active and passive monitoring, Appu presents a complete picture of information spread to the user. Additionally, using this knowledge of information spread, Appu will also classify user accounts into categories with varying security requirements. Then Appu suggests that the user strengthens the passwords on certain accounts if their category requires more protection due to personal information present in those accounts. Finally, Appu also monitors for personal information leakage over third party requests. Appu captures the implicit trust placed by the user on each site by monitoring the data submitted

by the user. Using this information, Appu automatically derives the correct security model ("security semantics") for user's online accounts. Based on this model, it encourages the user to change password with personalized warning messages ("security syntax").

In Chapter 4, we described the design and implementation of Newton, a browser extension that automatically finds web cookies required to access a specific part of a website. Furthermore, it presents the arrangement of these cookies in disjunctive normal form(DNF), which when satisfied, allows the HTTP request to succeed. Once this DNF equation of cookies is found, Newton checks whether any of the cookies in that equation can be stolen with common attacks such as man-in-the-middle or cross site scripting. Additionally, if insecure cookies are found from such a check, Newton also enforces protection flags on these cookies to resist session hijacking attempts. We found some errors in the cookie protection mechanism of 113 out of the Alexa top 200 sites. We reported these errors whenever possible, and got acknowledged for the same by site administrators. In two cases, as a result of our reporting, the developers of these sites changed their implementation. Newton automatically detects the auth-cookies ("security semantics") and protects them against session hijacking attacks by enforcing proper cookie flags ("security syntax").

In Chapter 5, we presented SilverLine, a data security framework to protect sensitive data in any cloud based web application. SilverLine requires minimal changes to the base web application. It automatically figures out the sensitive data flow by analyzing primary and foreign key relationships between tables. The administrator has to only specify this relationship at the start of the deployment and do minimal changes to application login process. We ported OSCommerce, a framework powering more than twelve thousand stores, to SilverLine and analyzed the performance overhead. We found that SilverLine imposes acceptable overhead in comparison to

the protection that it offers. SilverLine deduces the importance of the data ("security semantics"), track the data flow and automatically derive firewall rules ("security syntax") to stop data theft.

6.2 Future Directions

In this section, we discuss how the systems that we built can be improved as well as the next work that can be undertaken utilizing the currently built tools as base systems.

6.2.1 Personal Information Monitoring

Currently, when Appu is in passive monitoring mode, it will detect the user's personal information on a web page only when a user visits that page. The more time the user spends on the web service, the greater the likelihood of her visiting her profile setting page increases. However, a downside of this is when a user installs Appu, if no FPI is present for that web service, then the user will not know about all the personal information on that web service. A small search engine built into Appu can solve this problem. When a user logs into a service, the search engine would start following links which would link to her profile settings page. Machine learning techniques can be used to train this engine to find these links in an efficient way. For the majority of websites, the personal information can be found on a single page. However, if the information is scattered, the engine would have to explore the user's account. While doing this automated exploration, it is also important that Appu does not perform undesired actions - the least harmful of which could be logging the user out. However, more dangerous actions, such as deleting something from the user's account, or transferring money, must be avoided. For this reason, the search engine, in addition to being trained on what links should be followed, should also be trained with links strictly not to be followed with negative weights. Second, when users upload documents on websites, Appu should parse them and based on the contents, deduce the importance

of the document for that user. Obviously, if the document contains the user's real life name or address or other personal information, then such deduction is easy. However, inferring whether a chat message or a forum post contains something important can be hard. Some of the recent sentiment processing libraries could be useful in such cases. If the user has already uploaded documents on webservices such as Dropbox or Google Drive, then the question remains if Appu should parse such documents and if so, then how should Appu access them. The first question is easily solved by seeking the user's permission. However, the second question is hard since to be generic, Appu cannot rely on a particular site's API interfaces, as they will not work on another site. In addition to this, if Appu downloads these documents and analyzes them, then it would also consume network bandwidth and compute cycles. These are some of the interesting problems that can be worked on with Appu as the base system. Solving these challenges could help in building a personal search engine for user data behind the walls.

To build a third party leakage clearinghouse, Appu needs to improve leakage detection from low entropy personal information fields such as gender. This can possibly be achieved by analyzing the HTTP request context and analyzing other parameters. Also, anonymized reporting interface for reporting these leaks would have to be built carefully so that the clearinghouse itself cannot deanonymize the user from record linkages. The clearinghouse server should also support an API for other privacy researchers to download this information, analyze it and use it in their tools.

To detect an effective way to nudge the user towards "good online behavior," a wider study with more users would be helpful. In addition to detecting personal information spread, this study should also focus on getting detailed user experiences. Results of this study can be used in Appu to nudge the user in a most effective way and at an opportune time.

Finally, the collected information can be presented to the user so that the user is able to grasp the most important facts quickly. Libraries such as D3 can be highly effective in presenting this information. Additionally, a search interface would be a useful way to locate a specific fact quickly.

6.2.2 Session Hijacking

The current search and prune algorithm used in Newton explores the entire exponential search space. An engineering upgrade to this would be to implement the Apriori algorithm to explore the search space quickly. While evaluating Newton, we discovered that even on a single web service, different parts could have different authentication cookie set requirements to access them. Currently, we guessed these parts based on our knowledge about the service, or the change in subdomain, or the contents of the different parts. Since Newton is mainly designed for developers, a developer of the website is more effective at enlisting these logically distinct parts. However, if Newton has to run as a regression check, then it should not rely on the developer to provide this input. Also, if Newton is run by end users as a crowd sourcing tool, then it would have to figure out this information automatically. It is not possible for Newton to explore each and every part of the webservice to find the authentication cookies necessary to access it. To find logically distinct parts of the service automatically, more research needs to be done.

6.2.3 Cloud Security Framework

The administrator has to mark the sensitive tables in the database carefully for SilverLine to work properly. A misconfiguration such as a table with sensitive information being misclassified into a non-sensitive group, data leaks can still occur (although the leaks would be limited to data in that table). We believe that misconfiguration could be detected early, with test cases where a particular user's session leaks information about another user. Because our taint storage granularity applies to an entire row of

each table, a single test case per table should be suïňČcient enough to detect misconïňĄguration. With such a sanity test mechanism in place, the administrator can iteratively refine the configuration of SilverLine.

When using taint tracking for information flow control, one has to carefully choose the granularity of the taint tracking. Too detailed taint tracking gives accurate results, but adds performance overheads, making the system expensive; whereas too coarse of an approach causes taint explosion, and too many false positives. Since SilverLine mainly tracks the taints at the process level, it is ideal for web applications with strong isolation between users. However, to build SilverLine for applications where users heavily interact with each other requires rethinking about a single taint's meaning. Instead of treating the taint as a label, if the taint would be a pointer to a set of labels, then information sharing between the users can be handled. However, further performance study must be done to measure the performance overheads when a single user interacts with many others. This can be a cause of concern for applications exhibiting small world network relationships between users such as social networking applications.

To detect false negatives, approaches like formal verification and symbolic execution can be used. To avoid covert channel and side channel attacks, SilverLine would have to eliminate the common modulating channel in a similar way as done by other systems such as Determinator.

One of the engineering challenges for SilverLine is partial deployment, where an administrator can enable SilverLine for only a selected number of users. SilverLine needs to address how a subset of existing users would be isolated to enable such testing. A start in this direction could be the use of database sharding.

REFERENCES

[1] "';–have i been pwned?." https://goo.gl/l87IuE.

[2] "Blackmail, Deletion Offers Hit Ashley Madison Users." http://goo.gl/10gAT7.

[3] "Digg loses a third of its visitors in a month: is it deadd?." http://goo.gl/gZ3IkJ.

[4] "Edward Snowden: Leaks that exposed US spy programme." http://goo.gl/ghjhLf.

[5] "Fiber-Optic Internet In the United States." http://goo.gl/yQlxhX.

[6] "Firefox: Security/Tracking protection." https://goo.gl/YxDe1j.

[7] "FourthParty: Web Measurement Platform." http://goo.gl/Yhtid.

[8] "How the Ashley Madison Hack Could Threaten People's Lives." https://goo.gl/RNt4Ou.

[9] "Identity Theft Can Hurt You." http://goo.gl/8QC0CF.

[10] "Infidelity site Ashley Madison hacked as attackers demand total shutdown." http://goo.gl/plEjTv.

[11] "It's time to reveal the Israeli role in the US surveillance machine." http://goo.gl/jTTem4.

[12] "KrebsOnSecurity: PayIvy Sells Your Online Accounts Via PayPal." http://goo.gl/WCyR5I.

[13] "Lastpass sold to LogMeIn – should Linux users panic?." http://goo.gl/COfI7o.

[14] "McKinsey & Company: Offline and falling behind: Barriers to Internet adoption." http://goo.gl/0JLRCX.

[15] "Most people have heard of Snowden, few have changed habits as a result." http://goo.gl/4TqGjN.

[16] "MySpace lost 10 million users in a month; Close within the year?." http://goo.gl/iKgOfJ.

[17] "NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience!." https://goo.gl/9DktLp.

[18] "NoTrace PROTECT YOUR PRIVACY." http://goo.gl/rbmU3Q.

[19] "Number of monthly active Facebook users worldwide as of 2nd quarter 2015." http://goo.gl/8fzikj.

[20] "Selenium Web application testing system." http://www.seleniumhq.org.

[21] "Skyhigh: Cloud Adoption & Risk Report Q4 2015." https://goo.gl/9WNGg0.

[22] "Statista: Digital advertising spending worldwide from 2012 to 2018 (in billion U.S. dollars)." http://goo.gl/6ghv3W.

[23] "Techcrunch: Chromebook Sales Predicted To Grow 27% This Year, To 7.3M Units." http://goo.gl/Db59rW.

[24] "The Linkability of Usernames: a Step Towards âĂIJUber-ProfilesâĂİ." http://goo.gl/8YvG9k.

[25] "THERE IS NO SUCH THING AS ANONYMOUS ONLINE TRACKING." http://goo.gl/XK3ugk.

[26] "TRACKING THE TRACKERS: WHERE EVERYBODY KNOWS YOUR USERNAME." http://goo.gl/MqydO6.

[27] "World's Biggest Data Breaches." http://goo.gl/a0pGe.

[28] "World's Biggest Data Breaches." http://goo.gl/a0pGe.

[29] "XKCD: Password Reuse." https://goo.gl/rv3zWJ.

[30] "zxcvbn: realistic password strength estimation." http://goo.gl/hebzZ.

[31] (Washington, DC), Aug. 2001.

[32] Agrawal, R. and Srikant, R., "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, (San Francisco, CA, USA), pp. 487–499, Morgan Kaufmann Publishers Inc., 1994.

[33] "Amazon Relational Database Service." http://aws.amazon.com/rds/.

[34] "Unsafe cookies leave WordPress accounts open to hijacking, 2-factor bypass." Ars Technica. http://goo.gl/B1qLF7, May 2014.

[35] Atkins, D. and Austein, R., *Threat Analysis of the Domain Name System (DNS)*, Aug. 2004. RFC 3833.

[36] Attrition.net, "Absolute Sownage." http://attrition.org/security/rant/sony_aka_sownage.html, 2011.

[37] Barth, A., *The Web Origin Concept*, Dec. 2011. RFC 6454.

[38] "Bigcommerce: Ecommerce Software & Shopping Cart." https://www.bigcommerce.com/, Retrieved Feb. 2015.

[39] Bilge, L., Strufe, T., Balzarotti, D., and Kirda, E., "All your contacts are belong to us: automated identity theft attacks on social networks," in *Proceedings of the 18th international conference on World wide web*, pp. 551–560, ACM, 2009.

[40] Birnholtz, J., Merola, N. A. R., and Paul, A., "âĂIJis it weird to still be a virgin?:âĂİ anonymous, locally targeted questions on facebook confession boards," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pp. 2613–2622, ACM, 2015.

[41] Bonneau, J., "The science of guessing: Analyzing an anonymized corpus of 70 million passwords," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, (Washington, DC, USA), pp. 538–552, IEEE Computer Society, 2012.

[42] Bonneau, J., "Statistical metrics for individual password strength," in *Proceedings of the 20th International Conference on Security Protocols*, SP'12, (Berlin, Heidelberg), pp. 76–86, Springer-Verlag, 2012.

[43] Bonneau, J., Herley, C., Van Oorschot, P. C., and Stajano, F., "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 553–567, IEEE, 2012.

[44] Bugliesi, M., Calzavara, S., Focardi, R., and Khan, W., "Automatic and robust client-side protection for cookie-based sessions," in *International Symposium on Engineering Secure Software and Systems*, ESSoS'14, 2014.

[45] Burket, J., Mutchler, P., Weaver, M., Zaveri, M., and Evans, D., "Guardrails: A data-centric web application security framework," in *WebApps*, 2011.

[46] Business, V., "Verizon Data Breach Investigations Report." http://www.verizonbusiness.com/resources/reports/rp_data-breach-investigations-report-2011_en_xg.pdf, 2011.

[47] Butler, E., "A Firefox extension that demonstrates HTTP session hijacking attacks.." http://codebutler.github.io/firesheep/, 2010.

[48] Calzavara, S., Tolomei, G., Bugliesi, M., and Orlando, S., "Quite a mess in my cookie jar!: Leveraging machine learning to protect web authentication," in *Proceedings of the 23rd International Conference on World Wide Web*, 2014.

[49] Chiasson, S., Forget, A., Biddle, R., and van Oorschot, P. C., "Influencing users towards better passwords: Persuasive cued click-points," in *Proceedings of the 22Nd British HCI Group Annual Conference on People and Computers: Culture, Creativity, Interaction - Volume 1*, BCS-HCI '08, (Swinton, UK, UK), pp. 121–130, British Computer Society, 2008.

[50] Chiasson, S., Forget, A., Stobert, E., van Oorschot, P. C., and Biddle, R., "Multiple password interference in text passwords and click-based graphical passwords," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 500–511, ACM, 2009.

[51] Cohen, E. S. and Jefferson, D., "Protection in the hydra operating system," in *Symposium on Operating Systems Principles (SOSP)*, pp. 141–160, 1975.

[52] Dacosta, I., Chakradeo, S., Ahamad, M., and Traynor, P., "One-time cookies: Preventing session hijacking attacks with stateless authentication tokens," vol. 12, (New York, NY, USA), pp. 1:1–1:24, ACM, July 2012.

[53] Dalton, M., Zeldovich, N., and Kozyrakis, C., "Nemesis: Preventing authentication & access control vulnerabilities in web applications," in *18th Usenix Security Symposium*, 2009.

[54] Das, A., Bonneau, J., Caesar, M., Borisov, N., and Wang, X., "The tangled web of password reuse," in *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[55] Davis, B. and Chen, H., "Dbtaint: Cross-application information flow tracking via databases," in *Proceedings of the USENIX Conference on Web Applications*, June 2010.

[56] De Ryck, P., Nikiforakis, N., Desmet, L., Piessens, F., and Joosen, W., "Serene: Self-reliant client-side protection against session fixation," in *Proceedings of the 12th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, DAIS'12, (Berlin, Heidelberg), pp. 59–72, Springer-Verlag, 2012.

[57] Denning, D. E., "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, pp. 236–243, May 1976.

[58] Dignan, L., "Sony's data breach costs." `http://www.zdnet.com/blog/btl/sonys-data-breach-costs-likely-to-scream-higher/49161`, 2011.

[59] Enck, W., Gilbert, P., gon Chun, B., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N., "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Symposium on Operating Systems Principles (SOSP)*, Oct. 2010.

[60] Englehardt, S., Reisman, D., Eubank, C., Zimmerman, P., Mayer, J., Narayanan, A., and Felten, E. W., "Cookies that give you away: The surveillance implications of web tracking," in *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, (Republic and Canton of Geneva, Switzerland), pp. 289–299, International World Wide Web Conferences Steering Committee, 2015.

[61] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., *Hyptertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force, June 1999. RFC 2616.

[62] "Floodlight OpenFlow Controller." `http://floodlight.openflowhub.org/`.

[63] Florencio, D. and Herley, C., "A large-scale study of web password habits," in *Proceedings of the 16th international conference on World Wide Web*, pp. 657–666, ACM, 2007.

[64] Forget, A., Chiasson, S., van Oorschot, P. C., and Biddle, R., "Improving text passwords through persuasion," in *Proceedings of the 4th Symposium on Usable Privacy and Security*, SOUPS '08, (New York, NY, USA), pp. 1–12, ACM, 2008.

[65] Fu, K., Sit, E., Smith, K., and Feamster, N., "Dos and don'ts of client authentication on the Web," in *Proc. 10th USENIX Security Symposium* [31].

[66] "Hamster." `https://github.com/robertdavidgraham/hamster`, Retrieved Feb. 2015.

[67] Haque, S., Wright, M., and Scielzo, S., "A study of user password strategy for multiple accounts," in *Proceedings of the third ACM conference on Data and application security and privacy*, pp. 173–176, ACM, 2013.

[68] Hicks, B., Rueda, S., Jaeger, T., and McDaniel, P., "From Trusted to Secure: Building Applications that Enforce System Security," in *Proceedings of the USENIX Annual Technical Conference*, (Santa Clara, CA), June 2007.

[69] Hodges, J., Jackson, C., and Barth, A., *HTTP Strict Transport Security (HSTS)*, Nov. 2012. RFC 6797.

[70] "HttpOnly." `https://www.owasp.org/index.php/HttpOnly`, Retrieved Feb. 2015.

[71] InformationWeek, "Stanford Hospital Breach Exposes 20,000 ER Records." `http://www.informationweek.com/news/security/attacks/231601110`, 2011.

[72] Inquirer, T., "Citibank Hacked by altering URLs." `http://www.theinquirer.net/inquirer/news/2079431/citibank-hacked-altering-urls`, 2011.

[73] Irani, D., Webb, S., Li, K., and Pu, C., "Large online social footprints–an emerging threat," in *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 03*, CSE '09, (Washington, DC, USA), pp. 271–276, IEEE Computer Society, 2009.

[74] Jackson, C. and Barth, A., "Forcehttps: Protecting high-security web sites from network attacks," in *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, (New York, NY, USA), pp. 525–534, ACM, 2008.

[75] Jiang, X., Walters, A., Buchholz, F., Xu, D., Wang, Y.-M., and Spafford, E. H., "Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach," in *ICDCS*, June 2006.

[76] Jung, J., Sheth, A., Greenstein, B., Wetherall, D., and Gabriel Maganis, T. K., "Privacy Oracle: a System for Finding Application Leaks with Black Box Differential Testing," in *ACM Conference on Computer and Communications Security*, 2008.

[77] Kang, R., Brown, S., and Kiesler, S., "Why do people seek anonymity on the internet?: informing policy and design," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 2657–2666, ACM, 2013.

[78] Kang, R., Dabbish, L., Fruchter, N., and Kiesler, S., "âĂIJmy data just goes everywhere:âĂİ user mental models of the internet and implications for privacy and security," in *Symposium on Usable Privacy and Security (SOUPS)*, 2015.

[79] Kelley, P. G., Komanduri, S., Mazurek, M. L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., and Lopez, J., "Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 523–537, IEEE, 2012.

[80] Kohler, E., "Hot crap!," in *In Proceedings of WOWCS*, Apr. 2008.

[81] Komanduri, S., Shay, R., Kelley, P. G., Mazurek, M. L., Bauer, L., Christin, N., Cranor, L. F., and Egelman, S., "Of passwords and people: Measuring the effect of password-composition policies," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, (New York, NY, USA), pp. 2595–2604, ACM, 2011.

[82] Kranch, M. and Bonneau, J., "Upgrading https in midair: Hsts and key pinning in practice," in *NDSS '15: The 2015 Network and Distributed System Security Symposium*, February 2015.

[83] Krishnamurthy, B., "Privacy and online social networks: Can colorless green ideas sleep furiously?," *IEEE Security and Privacy*, vol. 11, pp. 14–20, May 2013.

[84] Krishnamurthy, B., Naryshkin, K., and Wills, C., "Privacy leakage vs. protection measures: the growing disconnect," in *Proceedings of the Web*, vol. 2, pp. 1–10, 2011.

[85] Krohn, M., Yip, A., Brodsky, M., Morris, R., and Walfish, M., "A World Wide Web Without Walls," in *6th ACM Workshop on Hot Topics in Networking (Hotnets)*, (Atlanta, GA), November 2007.

[86] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R., "Information flow control for standard os abstractions," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), pp. 321–334, ACM, 2007.

[87] Li, Z., He, W., Akhawe, D., and Song, D., "The emperor? s new password manager: Security analysis of web-based password managers," in *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association, 2014.

[88] Liu, Y., Gummadi, K. P., Krishnamurthy, B., and Mislove, A., "Analyzing facebook privacy settings: User expectations vs. reality," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, (New York, NY, USA), pp. 61–70, ACM, 2011.

[89] Liu, Y., Song, H. H., Bermudez, I., Mislove, A., Baldi, M., and Tongaonkar, A., "Identifying personal information in internet traffic," in *Proceedings of the 3rd ACM Conference on Online Social Networks (COSN'15)*, (Palo Alto, CA), November 2015.

[90] "Magento: Ecommerce Software & Ecommerce Platform." `http://magento.com/`, Retrieved Feb. 2015.

[91] Malandrino, D., Petta, A., Scarano, V., Serra, L., Spinelli, R., and Krishnamurthy, B., "Privacy awareness about information leakage: who knows what about me?," in *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pp. 279–284, ACM, 2013.

[92] Mayer, J. R. and Mitchell, J. C., "Third-party web tracking: Policy and technology," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 413–427, IEEE, 2012.

[93] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D., "Composing software-defined networks," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 2013.

[94] Myers, A. C., "Jflow: practical mostly-static information flow control," in *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 228–241, ACM, 1999.

[95] "MySQL Proxy." http://forge.mysql.com/wiki/MySQL_Proxy.

[96] Narayanan, A. and Shmatikov, V., "Fast dictionary attacks on passwords using time-space tradeoff," in *Proceedings of the 12th ACM conference on Computer and communications security*, pp. 364–372, ACM, 2005.

[97] "Nbtool." https://wiki.skullsecurity.org/Nbtool, Retrieved Feb. 2015.

[98] Newsome, J. and Song, D. X., "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), 2005.

[99] "Newton: Detailed Evaluation." https://goo.gl/6Lk0fw, June 2015.

[100] Nikiforakis, N., Meert, W., Younan, Y., Johns, M., and Joosen, W., "Sessionshield: Lightweight protection against session hijacking," in *Proceedings of the Third International Conference on Engineering Secure Software and Systems*, ESSoS'11, (Berlin, Heidelberg), pp. 87–100, Springer-Verlag, 2011.

[101] Notoatmodjo, G. and Thomborson, C., "Passwords and perceptions," in *Proceedings of the Seventh Australasian Conference on Information Security - Volume 98*, AISC '09, (Darlinghurst, Australia, Australia), pp. 71–78, Australian Computer Society, Inc., 2009.

[102] "OpenFlow Switch Consortium." http://www.openflowswitch.org/, 2008.

[103] "osCommerce: Open Source online shop e-commerce solution." http://www.oscommerce.com/.

[104] OWASP, "OWASP Top 10 Application Security Risks - 2010." https://www.owasp.org/index.php/Top_10_2010-Main, 2010.

[105] "Appu: FPI examples." https://goo.gl/qcllbR.

[106] "OWASP: Double Submit Cookies." http://goo.gl/qmW7o5, Retrieved Feb. 2015.

[107] "Testing for cookies attributes." https://www.owasp.org/index.php/Testing_for_cookies_attributes_(OWASP-SM-002), Retrieved Feb. 2015.

[108] Papagiannis, I., Migliavacca, M., and Pietzuch, P., "Php aspis: Using partial taint tracking to protect against injection attacks," in *2nd USENIX Conference on Web Application Development (WebApps)*, June 2011.

[109] Parno, B., McCune, J. M., Wendlandt, D., Andersen, D. G., and Perrig, A., "CLAMP: Practical prevention of large-scale data leaks," in *Proc. IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2009.

[110] P.Broadwell, M.Harren, N.Sastry, "Scrash: A system for generating security crash information," in *Proc. 12th USENIX Security Symposium*, (Washington, DC), Aug. 2003.

[111] Peddinti, S. T., Korolova, A., Bursztein, E., and Sampemane, G., "Cloak and swagger: Understanding data sensitivity through the lens of user anonymity," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 493–508, IEEE, 2014.

[112] Peddinti, S. T., Ross, K. W., and Cappos, J., "On the internet, nobody knows you're a dog: a twitter case study of anonymity in social networks," in *Proceedings of the second edition of the ACM conference on Online social networks*, pp. 83–94, ACM, 2014.

[113] Perito, D., Castelluccia, C., Kaafar, M. A., and Manils, P., "How unique and traceable are usernames?," in *Proceedings of the 11th International Conference on Privacy Enhancing Technologies*, PETS'11, (Berlin, Heidelberg), pp. 1–17, Springer-Verlag, 2011.

[114] Popa, R. A., Redfield, C. M. S., Zeldovich, N., and Balakrishnan, H., "Cryptdb: Protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 85–100, ACM, 2011.

[115] "POX: An OpenFlow controller." `http://www.noxrepo.org/pox/about-pox/`.

[116] `http://redis.io`.

[117] "Redis Benchmarks." `https://code.google.com/p/redis/wiki/Benchmarks`.

[118] Ren, J., Rao, A., Lindorfer, M., Legout, A., and Choffnes, D. R., "Recon: Revealing and controlling privacy leaks in mobile network traffic," *CoRR*, vol. abs/1507.00255, 2015.

[119] Roesner, F., Kohno, T., and Wetherall, D., "Detecting and defending against third-party tracking on the web.," in *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI 2012, 2012.

[120] "Seclist Advisory: Weak RNG in PHP session ID generation leads to session hijacking." `http://seclists.org/fulldisclosure/2010/Mar/519`, Retrieved Feb. 2015.

[121] "Session hijacking attack." `https://www.owasp.org/index.php/Session_hijacking_attack`, Retrieved Feb. 2015.

[122] Shay, R., Komanduri, S., Kelley, P. G., Leon, P. G., Mazurek, M. L., Bauer, L., Christin, N., and Cranor, L. F., "Encountering stronger password requirements: User attitudes and behaviors," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*, SOUPS '10, (New York, NY, USA), pp. 2:1–2:20, ACM, 2010.

[123] Silver, D., Jana, S., Chen, E., Jackson, C., and Boneh, D., "Password managers: Attacks and defenses," in *Proceedings of the 23rd Usenix Security Symposium*, 2014.

[124] "SSL Pulse: Survey of the SSL Implementation of the Most Popular Web Sites." `https://www.trustworthyinternet.org/ssl-pulse/`, Retrieved on Nov, 2014.

[125] Starov, O., Gill, P., and Nikiforakis, N., "Are you sure you want to contact us? quantifying the leakage of pii via website contact forms," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 1, pp. 20–33, 2016.

[126] Stobert, E., "The agony of passwords: Can we learn from user coping strategies?," in *CHI '14 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '14, (New York, NY, USA), pp. 975–980, ACM, 2014.

[127] Stuart, H. C., Dabbish, L., Kiesler, S., Kinnaird, P., and Kang, R., "Social transparency in networked information exchange: a theoretical framework," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pp. 451–460, ACM, 2012.

[128] Tang, S., Dautenhahn, N., and King, S. T., "Fortifying web-based applications automatically," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), pp. 615–626, ACM, 2011.

[129] Ur, B., Segreti, S. M., Bauer, L., Christin, N., Cranor, L. F., Komanduri, S., Kurilova, D., Mazurek, M. L., Melicher, W., and Shay, R., "Measuring real-world accuracies and biases in modeling password guessability," in *Proc. USENIX Security*, 2015.

[130] U.Shankar, K.Talwar, J.Foster, D.Wagner, "Detecting format string vulnerabilities with type qualifiers," in *Proc. 10th USENIX Security Symposium* [31].

[131] VanDeBogart, S., Efstathopoulos, P., Kohler, E., Krohn, M., Frey, C., Ziegler, D., Kaashoek, F., Morris, R., and Mazieres, D., "Labels and event processes in the Asbestos operating system," vol. 25, pp. 1–43, Dec. 2007.

[132] "Volusion Ecommerce Software & Shopping Cart Solutions." `http://www.volusion.com/`, Retrieved Feb. 2015.

[133] Walls, R. J., Clark, S. S., and Levine, B. N., "Functional privacy or why cookies are better with milk," in *Proceedings of the 7th USENIX Workshop on Hot Topics in Security*, HotSec '12, (Bellevue, WA), Aug. 2012.

[134] Wobber, E., Abadi, M., and Burrows, M., "Authentication in the Taos operating system," *ACM Transactions on Computer Systems*, vol. 12, no. 1, pp. 3–32, 1994.

[135] "WooCommerce - a free eCommerce toolkit for WordPress." http://www.woothemes.com/woocommerce/, Retrieved Feb. 2015.

[136] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G., "Linux Security Modules: General Security Support for the Linux Kernel," in *Proc. 11th USENIX Security Symposium*, (San Francisco, CA), Aug. 2002.

[137] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., and Wang, X. S., "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 1043–1054, ACM, 2013.

[138] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E., "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, (New York, NY, USA), pp. 116–127, ACM, 2007.

[139] Yip, A., Wang, X., Zeldovich, N., and Kaashoek, M. F., "Improving application security with data flow assertions," in *Proc. of the 22nd ACM Symposium on Operating System Principles*, Oct. 2009.

[140] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazieres, D., "Making Information Flow Explicit in HiStar," in *Proc. 7th USENIX OSDI*, (Seattle, WA), pp. 263–278, Nov. 2006.

[141] Zeldovich, N., Boyd-Wickizer, S., and Mazières, D., "Securing distributed systems with information flow control," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, (Berkeley, CA, USA), pp. 293–308, USENIX Association, 2008.

[142] Zhang, Q., McCullough, J., Ma, J., Schear, N., Vrable, M., Snoeren, A. C., Voelker, G. M., Savage, S., and Vahdat, A., "Neon: System Support for Derived Data Management," in *ACM Conference on Virtual Execution Environments*, Mar. 2010.